FREE 132 PAGE AMIGA E GUIDE

Amiga E

The new Amiga programming language

CU Amiga Magazine

Amiga E

Amiga E UK contact address:

Jason R Hulance
(Dept E)
Formal Systems (Europe) Ltd
3 Alfred Street
Oxford
OX1 4EH

Tel: 01869 324 350 (out of office hours)

E-mail: jason@fsel.com

Amiga E

A new and fast programming language for your Amiga.

This is a Beginner's Guide to Amiga E. It is designed to give you an introduction to the Amiga E programming language and, to some extent, programming in general.

Contents

Chapter 1

Chapter 2

Chapter 3

Chapter 4

Chapter 5

Chapter 6

Chapter 7

Chapter 8

Chapter 9

Chapter 10

Further Information

Chapter 1

1. Introduction to Amiga E

To interact with your Amiga you need to speak a language it understands. Luckily, there is a wide choice of such languages, each of which fits a particular need. For instance, BASIC (in most of its flavours) is simple and easy to learn, and so is ideal for beginners. Assembly, on the other hand, requires a lot of effort and is quite tedious, but can produce the fastest programs so is generally used by commercial programmers. These are two extremes and most businesses and colleges use C or Pascal/Modula-2, which try to strike a balance between simplicity and speed.

E programs look very much like Pascal or Modula-2 programs, but E is based more closely on C. Anyone familiar with these languages will easily learn E, only really needing to get to grips with E's unique features and those borrowed from other languages. This guide is aimed at people who haven't done much programming and may be too trivial for competent programmers. For those new to programming E, although dificult at first, will become clearer if you give it enough time and study this guide well.

Chapters 1-5 go through some of the basics of the E language and programming in general. Chapter 6 onwards delves deeper into E, covering the more complex topics and the unique features of E.

Note: The character '→' has been used throughout this guide to denote that a program line continues with no Return.

eg:
```
WriteF('My first→
program')
```
Should be typed as:
```
WriteF('My first program')
```

The reason we have had to do this is that, in some cases, lines are longer than our page width allows. Do NOT try to enter a character like → in your code. It will not work.

1.1 A Simple Program

If you're still reading you're probably desperate to do some programming in E but you don't know how to start. We'll therefore jump straight in the deep end with a small example. You'll need to know two things before we start: how to use a text editor and the Shell/CLI.

1.1.2 The code

Enter the following lines of code into a text editor and save it as the file 'simple.e' (taking care to copy each line accurately). (Just type the characters shown, and at the end of each line press the RETURN or ENTER key.)

```
PROC main()
  WriteF('My first program')
ENDPROC
```

Don't try to do anything different to the code, yet: the case of the letters in each word is significant and the funny characters are important. If you're a real beginner you might have difficulty finding the (') character. On my GB keyboard it's on the big key in the top left-hand corner directly below the ESC key. On a US and most European keyboards it's two to the right of the L key, next to the ; key.

1.1.3 Compilation

Once the file is saved (preferably in the RAM disk, since it's only a small program), you can use the E compiler to turn it into an executable program. All you need is the file 'ec' in your 'C:' directory or somewhere else on your search path (advanced users note: we don't need the 'Emodules:' assignment because we aren't using any modules). Assuming you have this and you have a Shell/CLI running, enter the following at the prompt after changing directory to where you saved your new file:

```
ec simple
```

If all's well you should be greeted, briefly, by the E compiler. If anything went wrong then double-check the contents of the file 'simple.e', that your CLI is in the same directory as this file, and that the program 'ec' is in your 'C:' directory (or on your search path).

1.1.4 Execution

Once everything is working you can run your first program by entering the following at the CLI prompt:

```
simple
```

As a help here's the complete transcript of the whole compilation and execution process (the CLI prompt, below, is the bit of text beginning with '1.' and ending in '>'):

```
1.Ram Disk:> ec simple
Amiga E Compiler/Assembler/Linker/PP v3.1a
registered (c) '91-95 Wouter lexical
analysing ... parsing and compiling ...
no errors
1.Ram Disk:> simple
My first program1.Ram Disk:>
```

Your display should be something similar if it's all worked. Notice how the output from the program runs into the prompt (the last line). We'll fix this soon.

Chapter 2

2. Understanding a Simple Program

To understand the example program we need to understand quite a few things. The observant amongst you will have noticed that all it does is print out a message, and that message was part of a line we wrote in the program. The first thing to do is see how to change this message.

2.1 Changing the Message

Edit the file so that line contains a different message between the two ' characters and compile it again using the same procedure as before. Don't use any ' characters except those around the message. If all went well, when you run the program again it should produce a different message. If something went wrong, compare the contents of your file with the original and make sure the only difference is the message between the ' characters.

2.1.1 Tinkering with the example

Simple tinkering is a good way to learn for yourself so it is encouraged on these simple examples. Don't stray too far, though, and if you start getting confused return to the proper example pretty sharpish!

2.1.2 Brief overview

We'll look in detail at the important parts of the program in the following sections, but we need first to get a glimpse of the whole picture. Here's a brief description of some fundamental concepts:

Procedures: We defined a procedure called 'main' and used the (built-in) procedure 'WriteF'. A procedure can be thought of as a small program with a name.

Parameters: The message in parentheses after 'WriteF' in our

program is the parameter to 'WriteF'. This is the data which the procedure should use.

Strings: The message we passed to 'WriteF' was a series of characters enclosed in ' characters. This is known as a "string".

2.2 Procedures

As mentioned above, a procedure can be thought of as a small program with a name. In fact, when an E program is run the procedure called 'main' is executed. Therefore, if your E program is going to do anything you must define a 'main' procedure. Other (built-in or user-defined) procedures may be run (or "called") from this procedure (as we did 'WriteF' in the example). For instance, if the procedure 'fred' calls the procedure 'barney' the code (or mini-program) associated with 'barney' is executed. This may involve calls to other procedures, and when the execution of this code is complete the next piece of code in the procedure 'fred' is executed (and this is generally the next line of the procedure). When the end of the procedure 'main' has been reached the program has finished. However, lots can happen between the beginning and end of a procedure, and sometimes the program may never get to finish.

Alternatively, the program may "crash", causing strange things to happen to your computer.

2.2.1 Procedure Definition

Procedures are defined using the keyword 'PROC', followed by the new procedure's name (starting with a lowercase letter), a description of the parameters it takes (in parentheses), a series of lines forming the code of the procedure and then the keyword 'ENDPROC'. Look at the example program again to identify the various parts.

2.2.2 Procedure Execution

Procedures can be called (or executed) from within the code part of another procedure. You do this by giving its name, followed by some data in parentheses. Look at the call to 'WriteF' in the example program.

2.2.3 Extending the example

Here's how we could change the example program to define another procedure:

```
PROC main()
   WriteF('My first program')
   fred()
ENDPROC

PROC fred()
   WriteF('...slightly improved')
ENDPROC
```

This may seem complicated, but in fact it's very simple. All we've done is define a second procedure called 'fred' which is just like the original program—it outputs a message. We've "called" this procedure in the 'main' procedure just after the line which outputs the original message. Therefore, the message in 'fred' is output after this message. Compile the program as before and run it so you don't have to take my word for it.

2.3 Parameters

Generally we want procedures to work with particular data. In our example we wanted the 'WriteF' procedure to work on a particular message. We passed the message as a "parameter" (or "argument") to 'WriteF' by putting it between the parentheses (the '(' and ')' characters) that follow the procedure name. When we called the 'fred' procedure, however, we did not require it to use any data so the parentheses were left empty.

When defining a procedure we define how much and what type of data we want it to work on, and when calling a procedure we give the specific data it should use. Notice that the procedure 'fred' (like the procedure 'main') has empty parentheses in its definition. This means that the procedure cannot be given any data as parameters when it is called. Before we can define our own procedure that takes parameters we must learn about variables. We'll do this in the next chapter.

2.4 Strings

A series of characters between two ' characters is known as a string. Almost any character can be used in a string, although the \ and ' characters have a special meaning. For instance, a linefeed is denoted by the two characters '\n'. We now know how to stop the message running into the prompt. Change the program to be:

```
PROC main()
   WriteF('My first program\n')
   fred()
ENDPROC

PROC fred()
   WriteF('...slightly improved\n')
ENDPROC
```

Compile it as before, and run it. You should notice that the messages now appear on lines by themselves, and the second message is separated from the prompt which follows it. We have therefore cured the linefeed problem we spotted earlier

2.5 Style, Reuse and Readability

The example has grown into two procedures, one called 'main' and one called 'fred'. However, we could get by with only one procedure:

```
PROC main()
   WriteF('My first program\n')
   WriteF('...slightly improved\n')
ENDPROC
```

What we've done is replace the call to the procedure 'fred' with the code it represents (this is called "inlining" the procedure). In fact, almost all programs can be easily re-written to eliminate all but the 'main' procedure. However, splitting a program up using procedures normally results in more readable code. It is also helpful to name your procedures so that their function is apparent, so our procedure 'fred' should probably have been named 'message' or something similar. A well-written program in this style can read just like English (or any other spoken language).

Another reason for having procedures is to reuse code, rather than having to write it out every time you use it. Imagine you wanted to print the same, long message fairly often in your program—you'd either have to write it all out every time, or you could write it once in a procedure and call this procedure when you wanted the message printed. Using a procedure also has the benefit of having only one copy of the message to change, should it ever need changing.

2.6 The Simple Program

The simple program should now (hopefully) seem simple. The only bit that hasn't been explained is the built-in procedure 'WriteF'. E has many built-in procedures and later we'll meet some of them in detail. The first thing we need to do, though, is manipulate data. This is really what a computer does all the time—it accepts data from some source (possibly the user), manipulates it in some way (possibly storing it somewhere, too) and outputs new data (usually to a screen or printer). The simple example program did all this, except the first two stages were rather trivial. You told the computer to execute the compiled program (this was some user input) and the real data (the message to be printed) was retrieved from the program. This data was manipulated by passing it as a parameter to 'WriteF', which then did some clever stuff to print it on the screen. To do our own manipulation of data we need to learn about variables and expressions.

Chapter 3

3. Variables and Expressions

Anybody who's done any school algebra will probably know what a variable is—it's just a named piece of data. In algebra the data is usually a number, but in E it can be all sorts of things (e.g., a string). The manipulation of data like the addition of two numbers is known as an "expression". The result of an expression can be used to build bigger expressions. For instance, '1+2' is an expression, and so is '6-(1+2)'. The good thing is you can use variables in place of data in expressions, so if 'x' represents the number 1 and 'y' represents 5, then the expression 'y-x' represents the number 4. In the next two sections we'll look at what kind of variables you can define and what the different sorts of expressions are.

3.1 Variables

Variables in E can hold many different kinds of data (called "types"). However, before a variable can be used it must be defined, and this is known as "declaring" the variable. A variable declaration also decides whether the variable is available for the whole program or just during the code of a procedure (i.e., whether the variable is "global" or "local"). Finally, the data stored in a variable can be changed using "assignments". The following sections discuss these topics in slightly more detail.

3.1.1 Variable types

In E a variable is a storage place for data (and this storage is part of the Amiga's RAM). Different kinds of data may require different amounts of storage. However, data can be

grouped together in "types", and two pieces of data from the same type require the same amount of storage. Every variable has an associated type and this dictates the maximum amount of storage it uses. Most commonly, variables in E store data from the type 'LONG'. This type contains the integers from -2,147,483,648 to 2,14C,3,647, so is normally more than sufficient. There are other types, such as 'INT' and 'LIST', and more complex things to do with types, but for now knowing about 'LONG' is enough.

3.1.2 Variable declaration

Variables must be declared before they can be used. They are declared using the 'DEF' keyword followed by a (comma-separated) list of the names of the variables to be declared. These variables will all have type 'LONG' (later we will see how to declare variables with other types). Some examples will hopefully make things clearer:

```
DEF x
DEF a, b, c
```

The first line declares the single variable 'x', whilst the second declares the variables 'a', 'b' and 'c' all in one go.

3.1.3 Assignment

The data stored by variables can be changed and this is normally done using "assignments". An assignment is formed using the variable's name and an expression denoting the new data it is to store. The symbol ':=' separates the variable from the expression. For example, the following code stores the number two in the variable 'x'. The left-hand side of the ':=' is the name of the variable to be affected ('x' in this case) and the right-hand side is an expression denoting the new value (simply the number two in this case).

```
x := 2
```

The following, more complex example uses the value stored in the variable before the assignment as part of the expression for the new data. The value of the expression on the right-hand side of the ':=' is the value stored in the variable 'x' plus one. This value is then stored in 'x', over-writing the previous data. (So, the overall effect is that 'x' is incremented.)

```
x := x + 1
```

This may be clearer in the next example which does not

change the data stored in 'x'. In fact, this piece of code is just a waste of CPU time, since all it does is look up the value stored in 'x' and store it back there!

```
x := x
```

3.1.4 Global and local variables (and procedure parameters)

There are two kinds of variable: "global" and "local". Data stored by global variables can be read and changed by all procedures, but data stored by local variables can only be accessed by the procedure to which they are local. Global variables must be declared before the first procedure definition. Local variables are declared within the procedure to which they are local (i.e., between the 'PROC' and 'ENDPROC'). For example, the following code declares a global variable 'w' and local variables 'x' and 'y'.

```
DEF w

PROC main()
DEF x
  x:=2
  w:=1
  fred()
ENDPROC

PROC fred()
DEF y
  y:=3
  w:=2
ENDPROC
```

The variable 'x' is local to the procedure 'main', and 'y' is local to 'fred'. The procedures 'main' and 'fred' can read and alter the value of the global variable 'w', but 'fred' cannot read or alter the value of 'x' (since that variable is local to 'main'). Similarly, 'main' cannot read or alter 'y'.

The local variables of one procedure are, therefore, completely different to the local variables of another procedure. For this reason they can share the same names without confusion. So, in the above example, the local variable 'y' in 'fred' could have been called 'x' and the program would have done exactly the same thing.

```
DEF w

PROC main()
DEF x
   x:=2
   w:=1
   fred()
ENDPROC

PROC fred()
DEF x
   x:=3
   w:=2
ENDPROC
```

This works because the 'x' in the assignment in 'fred' can refer only to the local variable 'x' of 'fred' (the 'x' in 'main' is local to 'main' so cannot be accessed from 'fred').

It a local variable for a procedure has the same name as a global variable then in the rest of the procedure the name refers only to the local variable. Therefore, the global variable cannot be accessed in the procedure, and this is called "descoping" the global variable.

The parameters of a procedure are local variables for that procedure. We've seen how to pass values as parameters when a procedure is called (the use of 'WriteF' in the example), but until now we haven't been able to define a procedure which takes parameters. Now we know a bit about variables we can have a go:

```
DEF y

PROC onemore(x)
   y:=x+1
ENDPROC
```

This isn't a complete program so don't try to compile it. Basically, we've declared a variable 'y' (which will be of type 'LONG') and a procedure 'onemore'. The procedure is defined with a parameter 'x', and this is just like a (local) variable declaration. When 'onemore' is called a parameter must be supplied, and this value is stored in the (local) variable 'x' before execution of 'onemore''s code. The code stores the value of 'x' plus one in the (global) variable 'y'. The following are some examples of calling 'onemore':

```
onemore(120)
onemore(52+34)
onemore(y)
```

A procedure can be defined to take any number of parameters. Below, the procedure 'addthem' is defined to take two parameters, 'a' and 'b', so it must therefore be called with two parameters. Notice that values stored by the parameter variables ('a' and 'b') can be changed within the code of the procedure, since they are just like local variables for the procedure. (The only real difference between local and parameter variables is that parameter variables are initialised with the values supplied as parameters when the procedure is called.

```
DEF y

PROC addthem(a, b)
   a:=a+2
   y:=a*b
ENDPROC
```

The following are some examples of calling 'addthem':

```
addthem(120,-20)
addthem(y,y+1)
```

Global variables are, by default, initialised to zero. Parameter variables are, of course, initialised by the actual values passed as parameters when a procedure is called. However, local variables are not initialised. This means that a local variable will contain a fairly random value when the code of a procedure is first executed. It is the responsibility of the programmer to ensure no assumptions are made about the value of local variables before they have been initialised. The obvious way to initialise a local variable is using an assignment, but there is also a way of giving an initialisation value as part of the declaration . Initialisation of variables is often very important, and is a common reason why programs go wrong.

3.1.5 Changing the example

Before we change the example we must learn something about 'WriteF'. We already know that the characters '\n' in a string mean a linefeed. However, there are several other important combinations of characters in a string, and some

are special to procedures like 'WriteF'. One such combination is '\d', which is easier to describe after we've seen the changed example.

```
PROC main()
   WriteF('My first program\n')
   fred()
ENDPROC

PROC fred()
   WriteF('...brought to you by the >
   number \d\n', 236)
ENDPROC
```

You might be able to guess what happens, but compile it and try it out anyway. If everything's worked you should see that the second message prints out the number that was passed as the second parameter to 'WriteF'. That's what the '\d' combination does—it marks the place in the string where the number should be printed.

We'll now try printing two numbers.

```
PROC main()
   WriteF('My first program\n')
   fred()
ENDPROC

PROC fred()
   WriteF('...brought to you by the numbers
   \d and \d\n', 16, 236)
ENDPROC
```

Because we're printing two numbers we need two lots of '\d', and we need to supply two numbers as parameters in the order in which we want them to be printed. The number 16 will therefore be printed before the word 'and' and before the number 236.

We can now make a big step forward and pass the numbers as parameters to the procedure 'fred'. Just look at the differences between this next example and the previous one.

```
PROC main()
   WriteF('My first program\n')
   fred(16, 236)
ENDPROC
```



```
PROC fred(a,b)
   WriteF('...brought to you by the >
   numbers\d and \d\n', a,b)
ENDPROC
```

This time we pass the (local) variables 'a' and 'b' to 'WriteF'. This is exactly the same as passing the values they store (which is what the previous example did), and so the output will be the same. In the next section we'll manipulate the variables by doing some arithmetic with 'a' and 'b', and get 'WriteF' to print the results.

3.2 Expressions

The E language includes the normal mathematical and logical operators. These operators are combined with values (usually in variables) to give "expressions" which yield new values. The following sections discuss this topic in more detail.

3.2.1 Mathematics

All the standard mathematical operators are supported in E. You can do addition, subtraction, multiplication and division. Other functions such as sine, modulus and square-root can also be used as they are part of the Amiga system libraries, but we only need to know about simple mathematics at the moment. The '+' character is used for addition, '-' for subtraction, '*' for multiplication (it's the closest you can get to a multiplication sign on a keyboard without using the letter 'x'), and '/' for division (be careful not to confuse the '\' used in strings with '/' used for division). The following are examples of expressions:

```
1+2+3+4
15-5
330/33
3*3+1
```

Each of these expressions yields ten as its result.
The last example is very carefully written to get the precedence correct.

All the above expressions use integer operators, so they manipulate integers, giving integers as results. "Floating-point" numbers are also supported by E, but using them is quite complicated and not discussed here. (Floating-point numbers can represent both very small fractions and very

large integers, but they have a limited accuracy, i.e. a limited number of "significant" digits.)

3.2.2 Logic and comparison

Logic lies at the very heart of a computer. They rarely guess what to do next; instead they rely on hard facts and precise reasoning. Consider the password protection on most games. The computer must decide whether you entered the correct number or word before it lets you play the game. When you play the game it's constantly making decisions: did your laser hit the alien?, have you got any lives left?, etc. Logic controls the operation of a program.

In E, the constants 'TRUE' and 'FALSE' represent the truth values true and false (respectively), and the operators 'AND' and 'OR' are the standard logic operators. The comparison operators are '=' (equal to), '>' (greater than), '<' (less than), '>=' (greater than or equal to), '<=' (less than or equal to) and '<>' (not equal to). All the following expressions are true:

```
TRUE
TRUE OR FALSE
1=1
3<>0
```

And these are all false:

```
FALSE
TRUE AND FALSE
0=2
(2<1) AND (-1=0)
```

The last example must use parentheses. We'll see why in the next section (it's to do with precedence, again).

The truth values 'TRUE' and 'FALSE' are actually numbers. This is how the logic system works in E. 'TRUE' is the number -1 and 'FALSE' is zero. The logic operators 'AND' and 'OR' expect such numbers as their parameters. In fact, the 'AND' and 'OR' operators are really bit-wise operators, so most of the time any non-zero number is taken to be 'TRUE'. It can sometimes be convenient to rely on this knowledge, although most of the time it is preferable (and more readable) to use a slightly more explicit form. Also, these facts can cause a few subtle problems as we shall see in the next section.

3.2.3 Precedence and grouping

At school most of us are taught that multiplications must be done before additions in a sum. In E it's different—there is no operator precedence, and the normal order in which the operations are performed is left-to-right, just like the expression is written. This means that expressions like '1+3*3' do not give the results a mathematician might expect. In fact, '1+3*3' represents the number 12 in E. This is because the addition, '1+3', is done before the multiplication, since it occurs before the multiplication. If the multiplication were written before the addition it would be done first (like we would normally expect).

Therefore, '3*3+1' represents the number 10 in E and in school mathematics.

To overcome this difference we can use parentheses to group the expression. If we'd written '1+(3*3)' the result would be 10. This is because we've forced E to do the multiplication first. Although this may seem troublesome to begin with, it's actually a lot better than learning a lot of rules for deciding which operator is done first (in C this can be a real pain, and you usually end up writing the brackets in just to be sure!).

The logic examples above contained the expression:

```
(2<1) AND (-1=0)
```

This expression was false. If we'd left the parentheses out, it would have been:

```
2<1 AND -1=0
```

This is actually interpreted the same as:

```
((2<1) AND -1) = 0
```

E calculates this (quite correctly) to be true. But the original expression (with parentheses) was false. Bracketing is therefore very important! It is also very easy to do correctly.

Chapter 4

4. Program Flow Control

A computer program often needs to repeatedly execute a series of statements or execute different statements according to the result of some decision. For example, a program to print all the numbers between one and a thousand would be very long and tedious to write if each print statement had to be given individually—it would be much better to use a variable and repeatedly print its value and increment it.

Another aspect of flow control is choosing between different pieces of code to execute. For instance, if something goes wrong a program may need to decide whether to continue or print an error message and stop—this part of a program is a typical example of a conditional block.

4.1 Conditional Block

There are two kinds of conditional block: 'IF' and 'SELECT'. Examples of these blocks are given below as fragments of E code (i.e., the examples are not complete E programs).

```
IF x>0
  x:=x+1
  WriteF('Increment: x is now \d\n', x)
ELSEIF x<0
  x:=x-1
  WriteF('Decrement: x is now \d\n', x)
ELSE
  WriteF('Zero: x is 0\n')
ENDIF
```

In the above 'IF' block, the first part checks if the value of 'x' is greater than zero, and, if it is, 'x' is incremented and the new value is printed (with a message saying it was incre-

mented). The program will then skip the rest of the block, and will execute the statements which follow the 'ENDIF'. If, however, 'x' it is not greater than zero the 'ELSEIF' part is checked, so if 'x' is less than zero it will be decremented and printed, and the rest of the block is skipped. If 'x' is not greater than zero and not less than zero the statements in the 'ELSE' part are executed, so a message saying 'x' is zero is printed. The 'IF' conditional is described in more detail below.

```
SELECT x
  CASE 0
    WriteF('x is zero\n')
  CASE 10
    WriteF('x is ten\n')
  CASE -2
    WriteF('x is -2\n')
  DEFAULT
    WriteF('x is not zero, ten or -2\n')
ENDSELECT
```

The 'SELECT' block is similar to the 'IF' block—it does different things depending on the value of 'x'. However, 'x' is only checked against specific values, given in the series of 'CASE' statements. If it is not any of these values the 'DEFAULT' part is executed.

There's also a variation on the 'SELECT' block (known as the 'SELECT..OF' block) which matches ranges of values and is quite fast. The two kinds of 'SELECT' block are described in more detail below.

4.1.1 'IF' block

The 'IF' block has the following form (the bits like <EXPRESSION> are descriptions of the kinds of E code which is allowed at that point—they are not proper E code):

```
IF <EXPRESSIONA>
  <STATEMENTSA>
ELSEIF <EXPRESSIONB>
  <STATEMENTSB>
ELSE
  <STATEMENTSC>
ENDIF
```


This block means:

- If <EXPRESSIONA> is true (i.e., represents 'TRUE' or any non-zero number) the code denoted by <STATEMENTSA> is executed.
- If <EXPRESSIONA> is false (i.e., represents 'FALSE' or zero) and <EXPRESSIONB> is true the <STATEMENTSB> part is executed.
- If both <EXPRESSIONA> and <EXPRESSIONB> are false the <STATEMENTSC> part is executed.

There does not need to be an 'ELSE' part but if one is present it must be the last part (immediately before the 'ENDIF'). Also, there can be any number of 'ELSEIF' parts between the 'IF' and 'ELSE' parts.

An alternative to this vertical form (where each part is on a separate line) is the horizontal form:

```
IF <EXPRESSION> THEN <STATEMENTA> ELSE
  <STATEMENTB>
```

This has the disadvantage of no 'ELSEIF' parts and having to cram everything onto a single line. Notice the presence of the 'THEN' keyword to separate the <EXPRESSION> and <STATEMENT>. This horizontal form is closely related to the 'IF' expression, which is described below To help make things clearer here are a number of E code fragments which illustrate the allowable 'IF' blocks:

```
IF x>0 THEN x:=x+1 ELSE x:=0
IF x>0
  x:=x+1
ELSE
  x:=0
ENDIF

IF x=0 THEN WriteF('x is zero\n')
IF x=0
  WriteF('x is zero\n')
ENDIF

IF x<0
  Write('Negative x\n')
ELSEIF x>2000
  Write('Too big x\n')
ELSEIF (x=2000) OR (x=0)
```

```
   Write('Worrying x\n')
ENDIF

IF x>0
IF x>2000
   WriteF('Big x\n')
ELSE
   WriteF('OK x\n')
ENDIF

ELSE
   IF x<-800 THEN WriteF('Small x\n') ->
   ELSE Write('Negative OK x')
ENDIF
```

In the last example there are "nested" 'IF' blocks (i.e., an 'IF' block within an 'IF' block). There is no ambiguity in which 'ELSE' or 'ELSEIF' parts belong to which 'IF' block because the beginning and end of the 'IF' blocks are clearly marked. For instance, the first 'ELSE' line can be interpreted only as being part of the innermost 'IF' block.

As a matter of style the conditions on the 'IF' and 'ELSEIF' parts should not "overlap" (i.e., at most one of the conditions should be true). If they do, however, the first one will take precedence. Therefore, the following two fragments of E code do the same thing:

```
IF x>0
   WriteF('x is bigger than zero\n')
ELSEIF x>200
   WriteF('x is bigger than 200\n')
ELSE
   WriteF('x is too small\n')
ENDIF

IF x>0
   WriteF('x is bigger than zero\n')
ELSE
   WriteF('x is too small\n')
ENDIF
```

The 'ELSEIF' part of the first fragment checks whether 'x' is greater than 200. But, if it is, the check in the 'IF' part would have been true ('x' is certainly greater than zero if it's greater than 200), and so only the code in the 'IF' part is executed. The whole 'IF' block behaves as if the 'ELSEIF' was not there.

4.1.2 'IF' expression

'IF' is such a commonly used construction that there is also an 'IF' expression. The 'IF' block is a statement and it controls which lines of code are executed, whereas the 'IF' expression is an expression and it controls its own value. For example, the following 'IF' block:

```
IF x>0
   y:=x+1
ELSE
   y:=0
ENDIF
```

can be written more succinctly using an 'IF' expression:

```
y:=(IF x>0 THEN x+1 ELSE 0)
```

The parentheses are unnecessary but they help to make the example more readable. Since the 'IF' block is just choosing between two assignments to 'y' it isn't really the lines of code that are different (they are both assignments), rather it is the values that are assigned to 'y' that are different. The 'IF' expression makes this similarity very clear. It chooses the "value" to be assigned in just the same way that the 'IF' block choose the "assignment".

The 'IF' expression has the following form:

```
IF <EXP> THEN <EXPA> ELSE <EXPB>
```

As you can see, 'IF' expressions are written like the horizontal form of the 'IF' block. However, there must be an 'ELSE' part and there can be no 'ELSEIF' parts. This means that the expression will always have a value (either <EXPA> or <EXPB>, depending on the value of <EXP>), and it isn't cluttered with lots of cases.

Don't worry too much about 'IF' expressions, since they are only useful in a handful of cases and can always be rewritten as a more wordy 'IF' block. Having said that they are very elegant and a lot more readable than the equivalent 'IF' block.

4.1.3 'SELECT' block

The 'SELECT' block has the following form:

```
SELECT <VARIABLE>
```

```
CASE <EXPRESSIONA>
   <STATEMENTSA>
CASE <EXPRESSIONB>
   <STATEMENTSB>
DEFAULT
   <STATEMENTSC>
ENDSELECT
```

The value of the selection variable (denoted by <VARIABLE> in the 'SELECT' part) is compared with the value of the expression in each of the 'CASE' parts in turn. If there's a match, the statements in the (first) matching 'CASE' part are executed. There can be any number of 'CASE' parts between the 'SELECT' and 'DEFAULT' parts. If there is no match, the statements in the 'DEFAULT' part are executed. There does not need to be a 'DEFAULT' part but if one is present it must be the last part (immediately before the 'ENDSELECT').

It should be clear that 'SELECT' blocks can be rewritten as 'IF' blocks, with the checks on the 'IF' and 'ELSEIF' parts being equality checks on the selection variable. For example, the following code fragments are equivalent:

```
SELECT x
CASE 22
   WriteF('x is 22\n')
CASE (y+z)/2
   WriteF('x is (y+x)/2\n')
DEFAULT
   WriteF('x isn't anything significant\n')
ENDSELECT

IF x=22
   WriteF('x is 22\n')
ELSEIF x=(y+z)/2
   WriteF('x is (y+x)/2\n')
ELSE
   WriteF('x isn't anything significant\n')
ENDIF
```

Notice that the 'IF' and 'ELSEIF' parts come from the 'CASE' parts, the 'ELSE' part comes from the 'DEFAULT' part, and the order of the parts is preserved. The advantage of the 'SELECT' block is that it's much easier to see that the value of 'x' is being tested all the time, and also we don't have to keep

writing 'x=' in the checks.

4.1.4 'SELECT..OF' block

The 'SELECT..OF' block is a bit more complicated than the normal 'SELECT' block, but can be very useful. It has the following form:

```
SELECT <MAXRANGE> OF <EXPRESSION>
CASE <CONSTA>
   <STATEMENTSA>
CASE <CONSTB1> TO <CONSTB2>
   <STATEMENTSB>
CASE <RANGE1>, <RANGE2>
   <STATEMENTSC>
DEFAULT
   <STATEMENTSD>
ENDSELECT
```

The value to be matched is <EXPRESSION>, which can be any expression, not just a variable like in the normal 'SELECT' block. However, the <MAXRANGE>, <CONSTA>, <CONSTB1> and <CONSTB2> must all be explicit numbers, i.e. constants. <MAXRANGE> must be a positive constant and the other constants must all be between zero and <MAXRANGE> (including zero but excluding <MAXRANGE>).

The 'CASE' values to be matched are specified using "ranges". A simple range is a single constant (the first 'CASE' above). The more general range is shown in the second 'CASE', using the 'TO' keyword (<CONSTB2> must be greater than <CONSTB1>). A general 'CASE' in the 'SELECT..OF' block can specify a number of possible ranges to match against by separating each range with a comma, as in the third 'CASE' above. For example, the following 'CASE' lines are equivalent and can be used to match any number from one to five (inclusive):

```
CASE 1 TO 5

CASE 1, 2, 3, 4, 5

CASE 1, 2 TO 3, 4 TO 5
```

If the value of the <EXPRESSION> is less than zero, greater than or equal to <MAXRANGE>, or it does not match any of

the constants in the 'CASE' ranges, then the statements in the 'DEFAULT' part are executed. Otherwise the statements in the first matching 'CASE' part are executed. As in the normal 'SELECT' block, there does not need to be a 'DEFAULT' part.

The following 'SELECT..OF' block prints the (numeric) day of the month nicely:

```
SELECT 32 OF day
CASE 1, 21, 31
   WriteF('The \dst day of the month\n', day)
CASE 2, 22
   WriteF('The \dnd day of the month\n', day)
CASE 3, 23
   WriteF('The \drd day of the month\n', day)
CASE 4 TO 20, 24 TO 30
   WriteF('The \dth day of the month\n', day)
DEFAULT
   WriteF('Error: invalid day=\d\n', day)
ENDSELECT
```

The <MAXRANGE> for this block is 32, since 31 is the maximum of the values used in the 'CASE' parts. If the value of 'day' was 100, for instance, then the statements in the 'DEFAULT' part would be executed, signalling an invalid day. This example can be rewritten as an 'IF' block:

```
IF (day=1) OR (day=21) OR (day=31)
   WriteF('The \dst day of the month\n', day)
ELSEIF (day=2) OR (day=22)
   WriteF('The \dnd day of the month\n', day)
ELSEIF (day=3) OR (day=23)
   WriteF('The \drd day of the month\n', day)
ELSEIF ((4<=day) AND (day<=20))   OR
   ((24<=day) AND (day<=30))  WriteF('The >
   \dth day of the month\n', day)
ELSE
   WriteF('Error: invalid day=\d\n', day)
ENDIF
```

The comma separating two ranges in the 'CASE' part has been replaced by an 'OR' of two comparison expressions, and the 'TO' range has been replaced by an 'AND' of two comparisons. (It is worth noticing the careful bracketing of the resulting expressions.)

Clearly, the 'SELECT..OF' block is much more readable than the equivalent 'IF' block. It is also a lot faster, mainly because none of the comparisons present in 'IF' block have to be done in the 'SELECT..OF' version. Instead the value to be matched is used to immediately locate the correct 'CASE' part. However, it's not all good news: the <MAXRANGE> value directly affects the size of compiled executable, so it is recommended that 'SELECT..OF' blocks be used only with small <MAXRANGE> values. See the 'Reference Manual' for more details.

4.2 Loops

Loops are all about making a program execute a series of statements over and over again. Probably the simplest loop to understand is the 'FOR' loop. There are other kinds of loops, but they are easier to understand once we know how to use a 'FOR' loop.

4.2.1 'FOR' loop

If you want to write a program to print the numbers one to 100 you can either type each number and wear out your fingers, or you can use a single variable and a small 'FOR' loop. Try compiling this E program (the space after the '\d' in the string is needed to separate the printed numbers):

```
PROC main()
   DEF x
FOR x:=1 TO 100
   WriteF('\d ', x)
ENDFOR
   WriteF('\n')
ENDPROC
```

When you run this you'll get all the numbers from one to 100 printed, just like we wanted. It works by using the (local) variable 'x' to hold the number to be printed. The 'FOR' loop starts off by setting the value of 'x' to one (the bit that looks like an assignment). Then the statements between the 'FOR' and 'ENDFOR' lines are executed (so the value of 'x' gets printed). When the program reaches the 'ENDFOR' it increments 'x' and checks to see if it is bigger than 100 (the limit we set with the 'TO' part). If it is, the loop is finished and the statements after the 'ENDFOR' are executed. If, however, it wasn't bigger than 100, the statements between the 'FOR' and

'ENDFOR' lines are executed all over again, and this time 'x' is one bigger since it has been incremented. In fact, this program does exactly the same as the following program (the '...' is not E code—it stands for the 97 other 'WriteF' statements):

```
PROC main()
  WriteF('\d ', 1)
  WriteF('\d ', 2)

  WriteF('\d ', 100)
  WriteF('\n')
ENDPROC
```

The general form of the 'FOR' loop is as follows:

```
FOR <VAR> := <EXPRESSIONA> TO <EXPRESSIONB>
STEP <NUMBER>
  <STATEMENTS>
ENDFOR
```

The <VAR> bit stands for the loop variable (in the example above this was 'x'). The <EXPRESSIONA> bit gives the start value for the loop variable and the <EXPRESSIONB> bit gives the last allowable value for it. The 'STEP' part allows you to specify the value (given by <NUMBER>) which is added to the loop variable on each loop. Unlike the values given for the start and end (which can be arbitrary expressions), the 'STEP' value must be a constant. The 'STEP' value defaults to one if the 'STEP' part is omitted (as in our example). Negative 'STEP' values are allowed, but in this case the check used at the end of each loop is whether the loop variable is "less than" the value in the 'TO' part. Zero is not allowed as the 'STEP' value.

As with the 'IF' block there is a horizontal form of a 'FOR' loop:

```
FOR <VAR> := <EXPA> TO <EXPB> STEP <EXPC>
DO <STATEMENT>
```

4.2.2 'WHILE' loop

The 'FOR' loop used a loop variable and checked whether that variable had gone past its limit. A 'WHILE' loop allows you to specify your own loop check. For instance, this program does the same as the program in the previous section:

```
PROC main()
  DEF x
    x:=1
  WHILE x<=100
    WriteF('\d ', x)
    x:=x+1
  ENDWHILE
    WriteF('\n')
  ENDPROC
```

We've replaced the 'FOR' loop with an initialisation of 'x' and a 'WHILE' loop with an extra statement to increment 'x'. We can now see the inner workings of the 'FOR' loop and, in fact, this is exactly how the 'FOR' loop works.

It is important to know that our check, 'x<=100', is done before the loop statements are executed. This means that the loop statements might not even be executed once. For instance, if we'd made the check 'x>=100' it would be false at the beginning of the loop (since 'x' is initialised to one in the assignment before the loop). Therefore, the loop would have terminated immediately and execution would pass straight to the statements after the 'ENDWHILE'.

Here's a more complicated example:

```
PROC main()
DEF x,y
  x:=1
  y:=2
WHILE (x<10) AND (y<10)
  WriteF('x is \d and y is \d\n', x, y)
  x:=x+2
  y:=y+2
ENDWHILE
ENDPROC
```

We've used two (local) variables this time. As soon as one of them is ten or more the loop is terminated. A bit of inspection of the code reveals that 'x' is initialised to one, and keeps having two added to it. It will, therefore, always be an odd number. Similarly, 'y' will always be even. The 'WHILE' check shows that it won't print any numbers which are greater than or equal to ten. From this and the fact that 'x' starts at one and 'y' at two we can decide that the last pair of numbers will be seven and eight. Run the program to confirm this.

Like the 'FOR' loop, there is a horizontal form of the 'WHILE' loop:

```
WHILE <EXPRESSION> DO <STATEMENT>
```

Loop termination is always a big problem. 'FOR' loops are guaranteed to eventually reach their limit (if you don't mess with the loop variable, that is). However, 'WHILE' loops (and all other loops) may go on forever and never terminate. For example, if the loop check were '1<2' it would always be true and nothing the loop could do would prevent it being true! You must therefore take care that your loops terminate in some way if you want to program to finish. There is a sneaky way of terminating loops using the 'JUMP' statement, but we'll ignore that for now.

4.2.3 'REPEAT..UNTIL' loop

A 'REPEAT..UNTIL' loop is very similar to a 'WHILE' loop. The only difference is where you specify the loop check, and when and how the check is performed. To illustrate this, here's the program from the previous two sections rewritten using a 'REPEAT..UNTIL' loop (try to spot the subtle differences):

```
PROC main()
  DEF x
  x:=1
REPEAT
  WriteF('\d ', x)
  x:=x+1
UNTIL x>100
  WriteF('\n')
ENDPROC
```

Just as in the 'WHILE' loop version we've got an initialisation of 'x' and an extra statement in the loop to increment 'x'. However, this time the loop check is specified at the end of the loop (in the 'UNTIL' part), and the check is only performed at the end of each loop. This difference means that the code in a 'REPEAT..UNTIL' loop will be executed at least once, whereas the code in a 'WHILE' loop may never be executed.

Also, the logical sense of the check follows the English: a 'REPEAT..UNTIL' loop executes "until" the check is true, whereas the 'WHILE' loop executes "while" the check is true. Therefore, the 'REPEAT..UNTIL' loop executes while the check is false! This may seem confusing at first, but just remember to read the code as if it were English and you'll get the correct interpretation.

Chapter 5

5. Summary

We've now completed the first 4 chapters, which was hopefully enough to get you started. If you've grasped the main concepts you are good position to proceed to the next few chapters, which cover the E language in more detail.

This is probably a good time to look at the different parts of one of the examples from the previous sections, since we've now used quite a bit of E. The following examination uses the 'WHILE' loop example. Just to make things are easier to follow, each line has been numbered (but don't try to compile it with the line numbers on!).

```
1. PROC main()
2.     DEF x,y
3.     x:=1
4.     y:=2
5. WHILE (x<10) AND (y<10)
6.     WriteF('x is \d and y is \d\n', x, y)
7.     x:=x+2
8.     y:=y+2
9.     ENDWHILE
10.ENDPROC
```

Hopefully, you should be able to recognise all the features listed in the above example using the table shown overleaf. If you don't then you might need to go back over the previous chapters. If you don't get it right up until this point you'll find it impossible to go further.

LINE(S)	OBSERVATION
1-10	The procedure definition.
1	The declaration of the procedure 'main', with no parameters.
2	The declaration of local variables 'x' and 'y'.
3, 4	Initialisation of 'x' and 'y' using assignment statements.
5-9	The 'WHILE' loop.
5	The loop check for the 'WHILE' loop using the logical operator 'AND', the comparison operator '<', and parentheses to group the expression.
6	The call to the (built-in) procedure 'WriteF' using parameters. Notice the string, the place holders for numbers, '\d', and the linefeed, '\n'.
7, 8	Assignments to 'x' and 'y', adding two to their values.
9	The marker for the end of the 'WHILE' loop.
10	The marker for the end of the procedure.

Chapter 6

6. Procedures and Functions

A "function" is a procedure which returns a value. This value can be formed from any expression so it may depend on the parameters with which the function was called. For instance, the addition operator '+' can be thought of as a function which returns the sum of its two parameters.

6.1 Functions

We can define our own addition function, 'add', in a very similar way to the definition of a procedure.

```
PROC main()
DEF sum
   sum:=12+79
   WriteF('Using +, sum is \d\n', sum)
   sum:=add(12,79)
   WriteF('Using add, sum is \d\n', sum)
ENDPROC

PROC add(x, y)
DEF s
   s:=x+y
ENDPROC s
```

This should generate the following output:

```
Using +, sum is 91
Using add, sum is 91
```

In the procedure 'add' the value 's' is returned using the 'ENDPROC' label. The value returned from 'add' can be used in expressions, just like any other value. You do this by writing the procedure call where you want the value to be. In the above example we wanted the value to be assigned to 'sum' so we wrote the call to 'add' on the right-hand side of

the assignment. Notice the similarities between the uses of '+' and 'add'. In general, 'add(a,b)' can be used in exactly the same places that 'a+b' can (more precisely, it can be used anywhere '(a+b)' can be used).

The 'RETURN' keyword can also be used to return values from a procedure. If the 'ENDPROC' method is used then the value is returned when the procedure reaches the end of its code. However, if the 'RETURN' method is used the value is returned immediately at that point and no more of the procedure's code is executed. Here's the same example using 'RETURN':

```
PROC add(x, y)
DEF s
   s:=x+y
RETURN s
ENDPROC
```

The only difference is that you can write 'RETURN' anywhere in the code part of a procedure and it finishes the execution of the procedure at that point (rather than execution finishing when it reaches the end of the code). In fact, you can use 'RETURN' in the 'main' procedure to prematurely finish the execution of a program.

```
PROC limitedadd(x,y)
   IF x>10000
RETURN 10000
   ELSEIF x<-10000
RETURN -10000
   ELSE
RETURN x+y
ENDIF
/* The following code is redundant */
x:=1
IF x=1 THEN RETURN 9999 ELSE RETURN -9999
ENDPROC
```

This function checks to see if 'x' is greater than 10,000 or less than -10,000, and if it is a limited value is returned (which is generally not the correct sum!). If 'x' is between -10,000 and 10,000 the correct answer is returned. The lines after the first 'IF' block will never get executed because execution will have finished at one of the 'RETURN' lines. Those lines are therefore just a waste of compiler time and can safely be omitted (as the comment suggests).

If no value is given with the 'ENDPROC' or 'RETURN' keyword then zero is returned. Therefore, all procedures are actually functions (and the terms "procedure" and "function" will tend to be used interchangeably). So, what happens to the value when you write a procedure call on a line by itself, not in an expression? Well, as we will see, the value is simply discarded. This is what happened in the previous examples when we called the procedures 'fred' and 'WriteF'.

6.2 One-Line Functions

Just as the 'IF' block and 'FOR' loop have horizontal, single line forms, so does a procedure definition. The general form is:

```
PROC <NAME> (<ARG1>, <ARG2>, ...) IS
   <EXPRESSION>
```

Alternatively, the 'RETURN' keyword can be used:

```
PROC <NAME> (<ARG1>, <ARG2>, ...) RETURN
   <EXPRESSION>
```

At first sight this might seem pretty unusable, but it is useful for very simple functions and our 'add' function in the previous section is a good example. If you look closely at the original definition you'll see that the local variable 's' wasn't really needed. Here's the one-line definition of 'add':

```
PROC add(x,y) IS x+y
```

6.3 Default Arguments

Sometimes a procedure (or function) will quite often be called with a particular (constant) value for one of its parameters, and it might be nice if you didn't have to fill this value in all the time. Luckily, E allows you to define "default" values for a procedure's parameters when you define the procedure. You can then just leave out that parameter when you call the procedure and it will default to the value you defined for it. Here's a simple example:

```
PROC play(track=1)
   WriteF('Starting to play track \d\n',
   track)
/* Rest of the code... */
ENDPROC
```

```
PROC main()
play(1)   -> Start playing from track 1
play(6)   -> Start playing from track 6
play() -> Start playing from track 1
ENDPROC
```

This is an outline of a program to control something like a CD player. The 'play' procedure has one parameter, 'track', which represents the first track that should be played. Often, though, you just tell the CD player to play, and don't specify a particular track. In this case, play starts from the first track. This is exactly what happens in the example above: the 'track' parameter has a default value of 1 defined for it (the '=1' in the definition of the 'play' procedure), and the third call to 'play' in 'main' does not specify a value for 'track', so the default value is used.

There are two constraints on the use of default arguments:

1. Any number of the parameters of a procedure may have default values defined for them, although they may only be the right-most parameters. This means that for a three parameter procedure, the second parameter can have a default value only if the last parameter does as well, and the first can have one only if both the others do. This should not be a big problem because you can always reorder the parameters in the procedure definition (and in all the places it has been called!).

The following examples show legal definitions of procedures with default arguments:

```
PROC fred(x, y, z) IS x+y+z -> No >
defaults

PROC fred(x, y, z=1) IS x+y+z -> z >
defaults to 1

PROC fred(x, y=23, z=1) IS x+y+z -> y >
and z have defaults
```

On the other hand, these definitions are all illegal:

```
PROC fred(x, y=23, z) IS x+y+z -> >
Illegal: no z default
PROC fred(x=9, y, z=1) IS x+y+z   ->
Illegal: no y default
```

2. When you call a procedure which has default arguments you can only leave out the right-most parameters. This means that for a three parameter procedure with all three parameters having default values, you can leave out the second parameter in a call to this procedure only if you also leave out the third parameter. The first parameter may be left out only if both the others are, too.

The following example shows which parameters are considered defaults:

```
PROC fred(x, y=23, z=1)
  WriteF('x is \d, y is \d, z is \d\n',
  x, y, z)
ENDPROC

PROC main()
  fred(2, 3, 4) -> No defaults used
  fred(2) -> y and z default
  fred() -> Illegal: x has no default
ENDPROC
```

In this example, you cannot leave out the 'y' parameter in a call to 'fred' without leaving out the 'z' parameter as well. To make 'y' have its default value and 'z' some value other than its default you need to supply the 'y' value explicitly in the call:

```
fred(2, 23, 9) -> Need to supply 23 for y
```

These constraints are necessary in order to make procedure calls unambiguous. Consider a three-parameter procedure with default values for two of the parameters. If it is called with only two parameters then, without these constraints, it would not be clear which two parameters had been supplied and which had not. If, however, the procedure were defined and called according to these constraints, then it must be the third parameter that needs to be defaulted (and the two parameters with default values must be the last two).

6.4 Multiple Return Values

So far we've only seen functions which return only one value, since this is something common to most programming

languages. However, E allows you to return up to three values from a function. To do this you list the values separated by commas after the 'ENDPROC', 'RETURN' or 'IS' keyword, where you would normally have specified only one value. A good example is a function which manipulates a screen coordinate, which is a pair of values: the x- and y-coordinates.

```
PROC movediag(x, y) IS x+8, y+4
```

All this function does is add 8 to the x-coordinate and 4 to the y-coordinate. To get to the return values other than the first one you must use a multiple-assignment statement:

```
PROC main()
  DEF a, b
  a, b:=movediag(10, 3)
  /* Now a should be 10+8, and b should >
  be 3+4 */
  WriteF('a is \d, b is \d\n', a, b)
ENDPROC
```

'a' is assigned the first return value and 'b' is assigned the second. You don't need to use all the return values from a function, so the assignment in the example above could have assigned only to 'a' (in which case it would not be a multiple-assignment anymore). A multiple-assignment makes sense only if the right-hand side is a function call, so don't expect things like the following example to set 'b' properly:

```
a,b:=6+movediag(10,3)    -> No obvious
value for b
```

If you use a function with more than one return value in any other expression (i.e., something which is not the right-hand side of an assignment), then only the first return value is used. For this reason the return values of a function have special names: the first return value is called the "regular" value of the function, and the other values are the "optional" values.

```
PROC main()
  DEF a, b
    /* The next two lines ignore the >
    second return value */
    a:=movediag(10, 3)
    WriteF('x-coord of movediag(21, 4) >
    is \d\n', movediag(21,4))
ENDPROC
```

Chapter 7

7. Constants

A "constant" is a value that does not change. A (literal) number like 121 is a good example of a constant—its value is always 121. We've already met another kind of constant: string constants. As you can doubtless tell, constants are pretty important things.

7.1 Numeric Constants

We've met a lot of numbers in the previous examples. Technically speaking, these were numeric constants (constant because they don't change value like a variable might). They were all decimal numbers, but you can use hexadecimal and binary numbers as well. There's also a way of specifying a number using characters. To specify a hexadecimal number you use a '$' before the digits (and after the optional minus sign '-' to represent a negative value). To specify a binary number you use a '%' instead.

Specifying numbers using characters is more complicated, because the base of this system is 256 (the base of decimal is ten, that of hexadecimal is 16 and that of binary is two). The digits are enclosed in double-quotes (the " character), and there can be at most four digits. Each digit is a character representing its ASCII value. Therefore, the character 'A' represents 65 and the character '0' (zero) represents 48. This upshot of this is that character 'A' has ASCII value '"A"' in E, and '"0z"' represents ("0" ★256) + "z" = (48★256) + 122 = 12,410. However, you probably don't need to worry about anything

other than the single character case, which gives you the ASCII value of the character.

The following table shows the decimal value of several numeric constants. Notice that you can use upper- or lower-case letters for the hexadecimal constants. Obviously the case of characters is significant for character numbers.

NUMBER	DECIMAL VALUE
21	21
-143	-143
$1a	26
-$B1	-177
%1110	14
-%1010	-10
"z"	122
"Je"	19,045
-"A"	-65

7.2 String Constants: Special Character Sequences

We have seen that in a string the character sequence '\n' means a linefeed. There are several other similar such special character sequences which represent useful characters that can't be typed in a string. The following table shows all these sequences. Note that there are some other similar sequences which are used to control formatting with built-in procedures like 'WriteF'. These are listed where 'WriteF' and similar procedures are described.

SEQUENCE	MEANING
\0	A null (ASCII zero)
\a	An apostrophe '
\b	A carriage return (ASCII 13)
\e	An escape (ASCII 27)
\n	A linefeed (ASCII 10)
\q	A double quote (ASCII 34)
\t	A tab (ASCII 9)
\\	A backslash \

An apostrophe can also be produced by typing two apostrophes in a row in a string. It's best to use this only in the middle of a string, where it's nice and obvious:

```
WriteF('Here''s an apostrophe.\n')
```

7.3 Named Constants

It is often nice to be able to give names to certain constants. For instance, as we saw earlier, the truth value 'TRUE' actually represents the value -1, and 'FALSE' represents zero . These are our first examples of named constants. To define your own you use the 'CONST' keyword as follows:

```
CONST ONE=1, LINEFEED=10, BIG_NUM=999999
```

This has defined the constant 'ONE' to represent one, 'LINEFEED' ten and 'BIG_NUM' 999,999. Named constants must begin with two uppercase letters.

You can use previously defined constants to give the value of a new constant, but in this case the definitions must occur on different 'CONST' lines.

```
CONST ZERO=0
CONST ONE=ZERO+1
```

The expression used to define the value of a constant can use only simple operators (no function calls) and constants.

7.4 Enumerations

Often you want to define a whole lot of constants and you just want them all to have a different value so you can tell them apart easily. For instance, if you wanted to define some constants to represent some famous cities and you only needed to know how to distinguish one from another then you could use an "enumeration" like this:

```
ENUM LONDON, MOSCOW, NEW_YORK, PARIS, ROME,
TOKYO
```

The 'ENUM' keyword begins the definitions (like the 'CONST' keyword does for an ordinary constant definition). The actual values of the constants start at zero and stretch up to five. In fact, this is exactly the same as writing:

```
CONST LONDON=0, MOSCOW=1, NEW_YORK=2,
PARIS=3, ROME=4, TOKYO=5
```

The enumeration does not have to start at zero, though. You

can change the starting value at any point by specifying a value for an enumerated constant. For example, the following constant definitions are equivalent:

```
ENUM APPLE, ORANGE, CAT=55, DOG, GOLDFISH

CONST APPLE=0, ORANGE=1, CAT=55, DOG=56, >
GOLDFISH=57
```

7.5 Sets

Yet another kind of constant definition is the "set" definition. This useful for defining flag sets, i.e., a number of options each of which can be on or off. The definition is like a simple enumeration, but using the 'SET' keyword and this time the values start at one and increase as powers of two (so the next value is two, the next is four, the next eight, and so on). Therefore, the following definitions are equivalent:

```
SET ENGLISH, FRENCH, GERMAN, JAPANESE, >
RUSSIAN

CONST ENGLISH=1, FRENCH=2, GERMAN=4, >
JAPANESE=8, RUSSIAN=16
```

However, the significance of the values it is best shown by using binary constants:

```
CONST ENGLISH=%00001, FRENCH=%00010, >
GERMAN=%00100, JAPANESE=%01000, >
RUSSIAN=%10000
```

If a person speaks just English then we can use the constant 'ENGLISH'. If they also spoke Japanese then to represent this with a single value we'd normally need a new constant (something like 'ENG_JAP'). In fact, we'd probably need a constant for each combination of languages a person might know. However, with the set definition we can 'OR' the 'ENGLISH' and 'JAPANESE' values together to get a new value, '%01001', and this represents a set containing both 'ENGLISH' and 'JAPANESE'. On the other hand, to find out if someone speaks French we would 'AND' the value for the languages they know with '%00010' (or the constant 'FRENCH'). (As you might have guessed, 'AND' and 'OR' are really bit-wise operators, not simply logical operators. Consider this program fragment:


```
speak:=GERMAN OR ENGLISH OR RUSSIAN
/* Speak any of these */
IF speak AND JAPANESE
      WriteF('Can speak in Japanese\n')
ELSE
      WriteF('Unable to speak in Japanese\n')
ENDIF
   IF speak AND (GERMAN OR FRENCH)
      WriteF('Can speak in German or >
      French\n')
ELSE
      WriteF('Unable to speak in German >
      or French\n')
ENDIF
```

The assignment sets 'speak' to show that the person can speak in German, English or Russian. The first 'IF' block tests whether the person can speak in Japanese, and the second tests whether they can speak in German or French.

When using sets be careful you don't get tempted to add values instead of 'OR'-ing them. Adding two different constants from the same set is the same as 'OR'-ing them, but adding the same set constant to itself isn't. This is not the only time addition doesn't give the same answer, but it's the most obvious. If you to stick to using 'OR' you won't have a problem.

Chapter 8

8. Types

We've already met the 'LONG' type and found that this was the normal type for variables. The types 'INT' and 'LIST' were also mentioned. Learning how to use types in an effective and readable way is very important. The type of a variable (as well as its name) can give clues to the reader about how or for what it is used. There are also more fundamental reasons for needing types, e.g., to logically group data using objects.

This is a very large chapter and you might like to take it slowly. One of the most important things to get to grips with is "pointers".

Concentrate on trying to understand these as they play a large part in any kind of system programming.

8.1 'LONG' Type

The 'LONG' type is the most important type because it is the default type and by far the most common type. It can be used to store a variety of data, including "memory addresses", as we shall see.

8.1.1 Default type

'LONG' is the default type of variables. It is a 32-bit type, meaning that 32-bits of memory (RAM) are used to store the data for each variable of this type and the data can take (integer) values in the range

-2,147,483,648 to 2,147,483,647. Variables default to being 'LONG' typed, but they can also be explicitly declared as 'LONG':

```
DEF x:LONG, y

PROC fred(p:LONG, q, r:LONG)
DEF zed:LONG
   <STATEMENTS>
ENDPROC
```

The global variable 'x', procedure parameters 'p' and 'r', and local variable 'zed' have all been declared to be 'LONG' values. The declarations are very similar to the kinds we've seen before, except that the variables have ':LONG' after their name in the declaration. This is the way the type of a variable is given. Note that the global variable 'y' and the procedure parameter 'q' are also 'LONG', since they do not have a type specified and 'LONG' is the default type for variables.

8.1.2 Memory addresses

There's a very good reason why 'LONG' is the normal type. A 32-bit (integer) value can be used as a "memory address". Therefore we can store the address (or location) of data in a variable (the variable is then called a "pointer"). The variable would then not contain the value of the data but a way of finding the data. Once the data location is known the data can be read or even altered! The next section covers pointers and addresses in more detail.

8.2 'PTR' Type

The 'PTR' type is used to hold memory addresses. Variables which have a 'PTR' type are called "pointers" (since they store memory addresses, as mentioned in the previous section). This section describes, in detail, addresses, pointers and the 'PTR' type.

8.2.1 Addresses

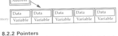
Every piece of data a program uses is stored somewhere in the computer's memory,and this includes the data contained in variables. So, when you assign one to the variable 'x' you are actually storing one in the location reserved for 'x' in the computer's memory. A location in memory is known as a "memory address", and this is just a 32-bit number (so can be stored in a 'LONG' variable). If you know the location of a variable's data then you can read the value and you can also

change it.

To understand memory addresses, a good analogy is to think of memory as a road or street, each memory location as a post-box on a house, and each piece of data as a letter. If you were a postman you would need to know where to put your letters, and this information is given by the address of the post-box. As time goes by, each post-box is filled with different letters. This is like the value in a memory location (or variable) changing. To change the letters stored in your post-box, you tell your friends your address and they can send letters in and fill it. This is like letting some part of a program change your data by giving it the address of the data.

The next two diagrams illustrate this analogy. A letter contains an address which points to a particular house (or lot of mail) on a street.

A pointer contains an address which points to a variable (or data) in memory.



8.2.2 Pointers

Variables which contain memory addresses are called "pointers". As we saw in the previous section, we can store memory addresses in 'LONG' variables. However, we then don't know the type of the data stored at those addresses. If it is important (or useful) to know this then the 'PTR' type (or, more accurately, one of the many 'PTR' types) should be used.

```
DEF p:PTR TO LONG, i:PTR TO INT,
    cptr:PTR TO CHAR, gptr:PTR TO gadget
```

The values stored in each of 'p', 'cptr', 'i' and 'gptr' are 'LONG' since they are memory addresses. However, the data at the address stored in 'p' is taken to be 'LONG' (a 32-bit value), that at 'cptr' is 'CHAR' (an 8-bit value), that at 'i' is 'INT' (a 16-bit value), and that at 'gptr' is 'gadget', which is an "object".

Since pointers are just data like any other 'LONG' variable, the value of the pointer is somewhere in memory. This means it has an address, so you can have a pointer which is actually pointing to another pointer! This is one of the reasons pointers can be quite difficult to think about, and misunderstanding them is often the cause of big problems with programs.

8.2.3 Indirect types

In the previous example we saw 'INT' and 'CHAR' used as the destination types of pointers, and these are the 16- and 8-bit equivalents (respectively) of the 'LONG' type. However, unlike 'LONG' these types cannot be used directly to declare global or local variables, or procedure parameters. They can only be used in constructing types (for instance with 'PTR TO'). The following declarations are therefore "illegal", and it might be nice to try compiling a little program with such a

```
/* This program fragment contains illegal
declarations */
DEF c:CHAR, i:INT
/* This program fragment contains illegal
declarations */
PROC fred(a:INT, b:CHAR)
DEF x:INT
  <STATEMENTS>
ENDPROC
```

This is not much of a limitation because you can store 'INT' or 'CHAR' values in 'LONG' variables if you really need to. However, it does mean there's a nice, simple rule: every direct value in E is a 32-bit quantity, either a 'LONG' or a pointer. In fact, 'LONG' is actually short-hand for 'PTR TO CHAR', so you can use 'LONG' values like they were actually 'PTR TO CHAR' values.

8.2.4 Finding addresses (making pointers)

If a program knows the address of a variable it can directly read or alter the value stored in the variable. To obtain the address of a simple variable you use '{' and '}' around the variable name. The address of non-simple variables (e.g., objects and arrays) can be found much more easily (see the appropriate section), and in fact you will very rarely need to use '{VAR}'. However, if you understand how to explicitly make pointers with '{VAR}' and use the pointers to get to data, then you'll understand the way pointers are used for the non-simple types much more quickly.

Addresses can be stored in a variable, passed to a procedure or whatever (they're just 32-bit values). Try out the following program:

```
DEF x

PROC main()
  fred(2)
ENDPROC

PROC fred(y)
  DEF z
    WriteF('x is at address \d\n', {x})
    WriteF('y is at address \d\n', {y})
    WriteF('z is at address \d\n', {z})
ENDPROC
```

This is an interesting program to run several times under different circumstances. You should see that sometimes the numbers for the addresses change. Running the program when another is multi-tasking (and eating memory) should produce the best changes, whereas running it consecutively (in one CLI) should produce the smallest (if any) changes. This gives you a glimpse at the complex memory handling of the Amiga and the E compiler.

8.2.5 Extracting data (dereferencing pointers)

If you have an address stored in a variable (i.e., a pointer) you can extract the data using the '^' operator. This is called "dereferencing" the pointer. The '^' operator should only really be used when '{VAR}' has been used to obtain an address. To this end, 'LONG' values are read and written when

dereferencing pointers in this way. For pointers to non-simple types (e.g., objects and arrays), dereferencing is achieved in much more readable ways (see the appropriate section for details), and this operator is not used. In fact, '^VAR' is seldom used in programs, but is useful for explaining how pointers work, especially in conjunction with '{VAR}'.

Using pointers can remove the scope restriction on local variables, i.e., they can be altered from outside the procedure for which they are local. Whilst this kind of use is not generally advised, it makes for a good example which shows the power of pointers. For example, the following program changes the value of the local variable 'x' for the procedure 'fred' from within the procedure 'barney'. It can do this only because 'fred' passes a pointer to 'x' as a parameter to 'barney'.

```
PROC main()
  fred()
ENDPROC

PROC fred()
  DEF x, p:PTR TO LONG
  x:=33
  p:={x}
  barney(p)
  WriteF('x is now \d\n', x)
ENDPROC

PROC barney(ptr:PTR TO LONG)
  DEF val
  val:=^ptr
  ^ptr:=val-6
ENDPROC
```

Notice that the '^' operator (i.e., dereferencing) is quite versatile. In the first assignment of the procedure 'barney' it is used (with the pointer 'ptr') to get the value stored in the local variable 'x', and in the second it is used to change this variable's value. In either case, dereferencing makes the pointer behave exactly as if you'd written the variable to which it points. To emphasise this, we can remove the 'barney' procedure, like we did above:

```
PROC main()
```



```
  fred()
ENDPROC

PROC fred()
  DEF x, p:PTR TO LONG, val
  x:=33
  p:={x}
  val:=x
  x:=val-6
  WriteF('x is now \d\n', x)
ENDPROC
```

Everywhere the 'barney' procedure used '^ptr' we've written 'x' (because we are now in the procedure for which 'x' is local). We've also eliminated the 'ptr' variable (the parameter to the 'barney' procedure), since it was only used with the '^' operator.

To make things clear the 'fred' and 'barney' example is deliberately 'wordy'. The 'val' and 'p' variables are unnecessary, and the pointer types could be abbreviated to 'LONG' or even omitted, for the reasons outlined above . This is the compact form of the example:

```
PROC main()
  fred()
ENDPROC

PROC fred()
  DEF x
  x:=33
  barney({x})
  WriteF('x is now \d\n', x)
ENDPROC

PROC barney(ptr)
  ^ptr:=^ptr-6
ENDPROC
```

By far the most common use of pointers is to address (or reference) large structures of data. It would be extremely expensive (in terms of CPU time) to pass large amounts of data from procedure to procedure, so addresses to such data are passed instead (and, as we know, these are just 32-bit values). The Amiga system functions (such as ones for creating

windows) require a lot of structured data, so if you plan to do any real programming you are going to have to understand and use pointers.

As we have seen, if you have a pointer to some data you can easily read the data, but you can just as easily alter it. If you want to write code that is clear and understandable, you have an implicit responsibility to not use pointers to alter data that you didn't ought to. For instance, if a procedure is passed a pointer which it then uses to change the data being pointed to, then it ought to be well documented (using comments) exactly what changes it makes.

8.2.6 Procedure parameters

Only local and global variables have the luxury of a large choice of types. Procedure parameters can only be 'LONG' or 'PTR TO TYPE'. This is not really a big limitation as we shall see in the later sections.

8.3 'ARRAY' Type

Quite often, the data used by a program needs to be ordered in some way, primarily so that it can be accessed easily. E provides a way to achieve such simple ordering: the 'ARRAY' type. This type (in its various forms) is common to most computer languages.

8.3.1 Tables of data

Data can be grouped together in many different ways, but probably the most common and straight-forward way is to make a table. In a table the data is ordered either vertically or horizontally, but the important thing is the relative positioning of the elements. The E view of this kind of ordered data is the 'ARRAY' type. An "array" is just a fixed sized collection of data in order. The size of an array is important and this is fixed when it is declared. The following illustrates array declarations:

```
DEF a[132]:ARRAY,
    table[21]:ARRAY OF LONG,
    ints[3]:ARRAY OF INT,
    objs[54]:ARRAY OF myobject
```

The size of the array is given in the square brackets ('[' and ']'). The type of the elements in the array defaults to 'CHAR',

but this can be given explicitly using the 'OF' keyword and the type name. However, only 'LONG', 'INT', 'CHAR' and object types are allowed ('LONG' can hold pointer values so this isn't much of a limitation). Object types are described below.

As mentioned above, procedure parameters cannot be arrays. We will overcome this apparent limitation soon.

8.3.2 Accessing array data

To access a particular element in an array you use square brackets again, this time specifying the "index" (or position) of the element you want. Indices start at zero for the first element of the array, one for the second element and, in general, (n-1) for the n-th element. This may seem strange at first, but it's the way most computer languages do it! We will see a reason why this makes sense soon (*note Array pointers::).

```
DEF a[10]:ARRAY

PROC main()
   DEF i
   FOR i:=0 TO 9
   a[i]:=i*i
ENDFOR
   WriteF('The 7th element of the array >
   a is \d\n', a[6])
   a[a[2]]:=10
   WriteF('The array is now:\n')
   FOR i:=0 TO 9
   WriteF(' a[\d] = \d\n', i, a[i])
ENDFOR
ENDPROC
```

This should all seem very straight-forward although one of the lines looks a bit complicated. Try to work out what happens to the array after the assignment immediately following the first 'WriteF'. In this assignment the index comes from a value stored in the array itself! Be careful when doing complicated things like this, though: make sure you don't try to read data from or write data to elements beyond the end of the

array. In our example there are only ten elements in the array 'a', so it wouldn't be sensible to talk about the eleventh element. The program could have checked that the value stored at 'a[2]' was a number between zero and nine before trying to access that array element, but it wasn't necessary in this case.

If you do try to write to a non-existent array element strange things can happen. This may be practically unnoticeable (like corrupting some other data), but if you're really unlucky you might crash your computer. The moral is: stay within the bounds of the array.

A short-hand for the first element of an array (i.e., the one with an index of zero) is to omit the index and write only the square brackets. Therefore, 'a[]' is the same as 'a[0]'.

8.3.3 Array pointers

When you declare an array, the address of the (beginning of the) array is given by the variable name without square brackets. Consider the following program:

```
DEF a[10]:ARRAY OF INT

PROC main()
   DEF ptr:PTR TO INT, i
   FOR i:=0 TO 9
      a[i]:=i
ENDFOR
   ptr:=a
   ptr++
   ptr[]:=22
FOR i:=0 TO 9
   WriteF('a[\d] is \d\n', i, a[i])
ENDFOR
ENDPROC
```

You should notice that the second element of the array has been changed using the pointer. The assignment to 'ptr' initialises it to point to the start of the array 'a', and then the 'ptr++' statement increments 'ptr' to point to the next element of the array. It is vital that 'ptr' is declared as 'PTR TO INT' since the array is an 'ARRAY OF INT'. The '[]' is used to dereference 'ptr' and therefore 22 is stored in the second element of the array. In fact, the 'ptr' can be used in exactly the same way as an array, so 'ptr[1]' would be the next (or third

element) of the array 'a'. Also, since 'ptr' points to the second element of 'a', negative values may legitimately be used as the index, and 'ptr[-1]' is the first element of 'a'.

In fact, the following declarations are identical except the first reserves an appropriate amount of memory for the array whereas the second relies on you having done this somewhere else in the program.

```
DEF a[20]:ARRAY OF INT

DEF a:PTR TO INT
```

The following diagram is similar to the diagrams given earlier (*note Addresses::). It is an illustration of an array, 'a', which was declared to be an array of twenty 'INT's.

As you can see, the variable 'a' is a pointer to the reserved chunk of memory which contains the array elements. Parts of memory that aren't between 'a[0]' and 'a[19]' are marked as 'Unknown' because they are not part of the array. This memory should therefore not be accessed using the array 'a'.

8.3.4 Point to other elements

We saw in the previous section how to increment a pointer so that it points to the next element in the array.

Decrementing a pointer 'p' (i.e., making it point to the previous element) is done in a similar way, using the 'p—' statement. Actually, 'p++' and 'p—' are really expressions which denote pointer values. 'p++' denotes the address stored in 'p' "before" it is incremented, and 'p—' denotes the address "after" 'p' is decremented. Therefore,

```
addr:=p
```

```
p++
```
does the same as
```
addr:=p++
```
And
```
p--
addr:=p
```
does the same as
```
addr:=p--
```
The reason why '++' and '--' should be used to increment and decrement a pointer is that values from different types occupy different numbers of memory locations. In fact, a single memory location is a "byte", and this is eight bits. therefore, 'CHAR' values occupy a single byte, whereas 'LONG' values take up four bytes (32 bits). If 'p' were a pointer to 'CHAR' and it was pointing to an array (of 'CHAR') the 'p+1' memory location would contain the second element of the array (and 'p+2' the third, etc.). But if 'p' were a pointer to an array of 'LONG' the second element in the array would be at 'p+4' (and the third at 'p+8'). The locations 'p', 'p+1', 'p+2' and 'p+3' all make up the 'LONG' value at address 'p'. Having to remember things like this is a pain, and it's a lot less readable than using '++' or '--'. However, you must remember to declare your pointer with the correct type in order for '++' and '--' to work correctly.

8.3.5 Array procedure parameters

Since we now know how to get the address of an array we can simulate passing an array as a procedure parameter by passing the address of the array. For example, the following program uses a procedure to fill in the first 'x' elements of an array with their index numbers.

```
DEF a[10]:ARRAY OF INT

PROC main()
   DEF i
   fillin(a, 10)
   FOR i:=0 TO 9
WriteF('a[\d] is \d\n', i, a[i])
   ENDFOR
ENDPROC
```



```
PROC fillin(ptr:PTR TO INT, x)
   DEF i
   FOR i:=0 TO x-1
   ptr[]:=i
   ptr++
ENDFOR
   ENDPROC
```
The array 'a' only has ten elements so we shouldn't fill in any more than the first ten elements. Therefore, in the example, the call to the procedure 'fillin' should not have a bigger number than ten as the second parameter. Also, we could treat 'ptr' more like an array (and not use '++'), but in this case using '++' is slightly better since we are assigning to each element in turn. The alternative definition of 'fillin' (without using '++') is:
```
PROC fillin2(ptr:PTR TO INT, x)
   DEF i
   FOR i:=0 TO x-1
   ptr[i]:=i
   ENDFOR
ENDPROC
```
Also, yet another version of 'fillin' uses the expression form of '++' in the assignment and the horizontal form of the 'FOR' loop to give a really compact definition.
```
PROC fillin3(ptr:PTR TO INT, x)
   DEF i
   FOR i:=0 TO x-1 DO ptr[]++:=i
ENDPROC
```

8.4 'OBJECT' Type

Objects are the E equivalent of C and Assembly structures, or Pascal records. They are like arrays except the elements are named not numbered, and they can be of different types. So, to find a particular element in an object you use a name instead of an index (number). Objects are also the basis of the OOP features of E.

8.4.1 Example object

We'll dive straight in with this first example, and define an object and use it. Object definitions are global and must be

made before any procedure definitions.

```
OBJECT rec
   tag, check
   table[8]:ARRAY
   data:LONG
ENDOBJECT

PROC main()
   DEF a:rec
   a.tag:=1
   a.check:=a
   a.data:=a.tag+(10000*a.tag)
ENDPROC
```

This program doesn't visibly do anything so there isn't much point in compiling it. What it does do, however, is show how a typical object is defined and how elements of an object are selected.

The object being defined in the example is 'rec', and its elements are defined just like variable declarations (but without a 'DEF'). There can be as many lines of element definitions as you like between the 'OBJECT' and 'ENDOBJECT' lines, and each line can contain any number of elements separated by commas. The elements of the 'rec' object are 'tag' and 'check' (which are 'LONG'), 'table' (which is an array of 'CHAR' with eight elements) and 'data' (which is also 'LONG'). Every variable of 'rec' object type will have space reserved for each of these elements. The declaration of the (local) variable 'a' therefore reserves enough memory for one 'rec' object.

8.4.2 Element selection and element types

To select elements in an object 'obj' you use 'obj.name', where 'name' is one of the element names. In the example, the 'tag' element of the 'rec' object 'a' is selected by writing 'a.tag'. The other elements are selected in a similar way.

Just like an array declaration the address of an object 'obj' is stored in the variable 'obj', and any pointer of type 'PTR TO OBJECTNAME' can be used just like an object of type OBJECTNAME. Therefore, in the previous example 'a' is a 'PTR TO rec'.

As the example object shows, the elements of an object can have several different types. In fact, the elements can have

any type, including object, pointer to object and array of object. The following example shows how to access some different typed elements.

```
OBJECT rec
   tag, check
   table[8]:ARRAY
   data:LONG
ENDOBJECT

OBJECT bigrec
   data:PTR TO LONG
   subrec:PTR TO rec
   rectable[22]:ARRAY OF rec
ENDOBJECT

PROC main()
   DEF r:rec, b:bigrec, rt:PTR TO rec
   r.table[]:="H"
   b.subrec:=r
   b.subrec.tag:=1
   b.subrec.table[1]:="i"
   b.rectable[0].table[0]:="A"
   rt:=b.rectable
   rt[].data++:=0
   rt[].table[]--:="B"
ENDPROC
```

The '++' and '--' operators apply to first thing in the selection (i.e., 'rt' in "both" the last two assignments in the example above), and may occur only after all the selections.
Notice that object selection and array indexing can be repeated as much as necessary (but only as the types of the elements allow). As a simple example, consider the third assignment:

```
b.subrec.tag:=1
```

This selects the 'subrec' element from the 'bigrec' object 'b', and then sets the 'tag' element of this 'rec' object to 1. Now, consider one of the later assignments:

```
b.rectable[0].table[0]:="A"
```

This selects the 'rectable' element from 'b', which is an array of 'rec' objects. The first element of this array is selected, and then the 'table' element of the 'rec' object is selected. Finally,

the first character of the 'table' is set to the character 'A'.

As you can probably tell, it is important to give the elements of objects appropriate types if you want to do multiple selection in this way. However, this is not always possible or the best way of doing some things, so there is a way of giving a different type to pointers (this is called "explicit pointer typing"—see the 'Reference Manual' for more details).

Here's a quite simple example which uses an array of objects:

```
OBJECT rec
   tag, check
   table[8]:ARRAY
   data:LONG
ENDOBJECT

PROC main()
   DEF a[10]:ARRAY OF rec, p:PTR TO rec, i
   p:=a
   FOR i:=0 TO 9
   a[i].tag:=i
   p.check++:=i
ENDFOR
FOR i:=0 TO 9
   IF a[i].tag<>a[i].check
   WriteF('Whoops, a[\d] went wrong...\n', i)
ENDIF
   ENDFOR
ENDPROC
```

If you think about it for long enough you'll see that 'a[0].tag' is the same as 'a.tag'. That's because 'a' is a pointer to the first element of the array, and the elements of the array are objects. Therefore, 'a' is a pointer to an object (the first object in the array).

8.4.3 Amiga system objects

There are many different Amiga system objects. For instance, there's one which contains the information needed to make a gadget (like the 'close' gadget on most windows), and one which contains all the information about a process or task. These objects are vitally important and so are supplied with E in the form of 'modules'. Each module is specific to a certain

area of the Amiga system and contains object and other definitions.

8.5 'LIST' and 'STRING' Types

Arrays are common to many computer languages. However, they can be a bit of a pain because you always need to make sure you haven't run off the end of the array when you're writing to it. This is where the 'STRING' and 'LIST' types come in. 'STRING' is very much like 'ARRAY OF CHAR' and 'LIST' is like 'ARRAY OF LONG'. However, each has a set of E (built-in) functions which safely manipulate variables of these types without exceeding their bounds.

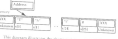
8.5.1 Normal strings and E-strings

"Normal" strings are common to most programming languages. They are simply an array of characters, with the end of the string marked by a null character (ASCII zero). We've already met normal strings. The ones we used were constant strings contained in ' characters, and they denote pointers to the memory where the string data is stored. Therefore, you can assign a string constant to a pointer (to 'CHAR'), and you've got a ready-filled array with the elements you want (an "initialised" array.)

```
DEF s:PTR TO CHAR
s:='This is a string constant'
/* Now s[] is T and s[2] is i */
```

Remember that 'LONG' is actually 'PTR TO CHAR' so this code is precisely the same as:

```
DEF s
s:='This is a string constant'
```



This diagram illustrates the above assignment to 's'. The first two characters 's[0]' and 's[1]') are 'T' and 'h', and the last

character (before the terminating null) is 't'. Memory marked as 'Unknown' is not part of the string constant.

"E-strings" are very similar to normal strings and, in fact, an E-string can be used wherever a normal string can. However, the reverse is not true, so if something requires an E-string you cannot use a normal string instead. The difference between a normal string and an E-string was hinted at in the introduction to this section: E-strings can be safely altered without exceeding their bounds. A normal string is just an array so you need to be careful not to exceed its bounds. However, an E-string knows what its bounds are, and so any of the string manipulation functions can alter them safely.

An E-string ('STRING' type) variable is declared as in the following example, with the maximum size of the E-string given just like an array declaration.

```
DEF s[30]:STRING
```

As with an array declaration, the variable 's' is actually a pointer to the string data. To initialise an E-string you need to use the function 'StrCopy' as we shall see.

8.5.2 String functions

There are a number of useful built-in functions which manipulate strings. Remember that an E-string can be used wherever a normal string can, but normal strings cannot be used where an E-string is required.If a parameter is marked as <STRING> then a normal or E-string can be passed as that parameter, but if it is marked as <E-STRING> then only an E-string may be used. Some of these functions have default arguments, which means you don't need to specify some parameters to get the default values . (You can, of course, ignore the defaults and always give all parameters.)

'String(<MAXSIZE>)' Allocates memory for an E-string of maximum size <MAXSIZE> and returns a pointer to the string data. It is used to make space for a new E-string, like a 'STRING' declaration does. The following code fragments are practically equivalent:

```
DEF s[37]:STRING

DEF s:PTR TO CHAR
s:=String(37)
```


The slight difference is that there may not be enough memory left to hold the E-string when the 'String' function is used. In that case the special value 'NIL' (a constant) is returned. Your program *must* check that the value returned is not 'NIL' before you use it as an E-string (or dereference it).

The memory for the declaration version, 'STRING', is allocated when the program is run, so your program won't run if there isn't enough memory. The 'String' version is often called "dynamic" allocation because it happens only when the program is running; the declaration version has allocation done by the E compiler.

The memory allocated using 'String' can be deallocated using 'DisposeLink'.

'StrCmp(<STRING1>,<STRING2>,<LENGTH>=ALL)'

Compares <STRING1> with <STRING2> (they can both be normal or E-strings). Returns 'TRUE' if the first <LENGTH> characters of the strings match, and 'FALSE' otherwise. The <LENGTH> defaults to the special constant 'ALL' which means that the strings must agree on every character. For example, the following comparisons all return 'TRUE':

```
StrCmp('ABC', 'ABC')
StrCmp('ABCd', 'ABC', 3)
```

And the following return 'FALSE' (notice the case of the letters):

```
StrCmp('ABC', 'ABc')
StrCmp('ABCd', 'ABC', ALL)
```

'StrCopy(<E-STRING>,<STRING>,<LENGTH>=ALL)'

Copies the contents of <STRING> to <E-STRING>, and also returns a pointer to the resulting E-string (for convenience). Only <LENGTH> characters are copied from the source string, but the special constant 'ALL' can be used to indicate that the whole of the source string is to be copied (and this is the default value for <LENGTH>). Remember that E-strings are safely manipulated, so the following code fragment results in 's' becoming 'More th', since its maximum size is (from its declaration) seven characters.

```
DEF s[7]:STRING
StrCopy(s, 'More than seven characters', ALL)
```

A declaration using 'STRING' (or 'ARRAY') reserves a small part of memory, and stores a pointer to this memory in the variable being declared. So to get data into this memory you need to copy it there, using 'StrCopy'. If you're familiar with very high-level languages like BASIC you should take care, because you might think you can assign a string to an array or an E-string variable.In E (and languages like C and Assembly) you must explicitly copy data into arrays and E-strings. You should not do the following:

```
/* You don't want to do things like this! */
DEF s[80]:STRING
s:='This is a string constant'
```

This is fairly disastrous: it throws away the pointer to reserved memory that was stored in 's' and replaces it by a pointer to the string constant.'s' is then no longer an E-string and *cannot* be repaired using 'SetStr'. If you want 's' to contain the above string you must use 'StrCopy':

```
DEF s[80]:STRING
StrCopy(s,'This is a string constant')
```

The moral is: remember when you are using pointers to data and when you need to copy data. Also, remember that assignment does not copy large arrays of data, it copies only pointers to data, so if you want to store some data in an 'ARRAY' or 'STRING' type variable you need to copy it there.

'StrAdd(<E-STRING>,<STRING>,<LENGTH>=ALL)'

This does the same as 'StrCopy' but the source string is copied onto the end of the destination E-string. The following code fragment results in 's' becoming 'This is a string and a half'.

```
DEF s[30]:STRING
StrCopy(s, 'This is a string', ALL)
StrAdd(s, ' and a half')
```

'StrLen(<STRING>)'

Returns the length of <STRING>. This assumes that the string is terminated by a null character (i.e., ASCII zero), which is true for any strings made from E-strings and string constants. However, you can make a string constant look short if you use the null character (the special sequence '\0') in it. For instance, these calls all return three:


```
StrLen('abc')
StrLen('abc\0def')
```

In fact, most of the string functions assume strings are null-terminated, so you shouldn't use null characters in your strings unless you really know what you're doing.
For E-strings 'StrLen' is less efficient than the 'EstrLen' function.

'EstrLen(<E-STRING>)'

Returns the length of <E-STRING> (remember this can be only an E-string). This is much more efficient than 'StrLen' since E-strings know their length and it doesn't need to search the string for a null character.

'StrMax(<E-STRING>)'

Returns the maximum length of <E-STRING>. This is not necessarily the current length of the E-string, rather it is the size used in the declaration with 'STRING' or the call to 'String'.

'RightStr(<E-STRING1>,<E STRING2>,<LENGTH>)'

This is like 'StrCopy' but it copies the right-most characters from <E-STRING2> to <E-STRING1> and both strings must be E-strings. At most <LENGTH> characters are copied, and the special constant 'ALL' *cannot* be used (to copy all the string you should, of course, use 'StrCopy'). For instance, a value of one for <LENGTH> means the last character of <E-STRING2> is copied to <E-STRING1>.

'MidStr(<E STRING>,<STRING>,<INDEX>,<LENGTH>=ALL)'

Copies the contents of <STRING> starting at <INDEX> (which is an index just like an array index) to <E-STRING>. At most <LENGTH> characters are copied, and the special constant 'ALL' can be used if all the remaining characters in <STRING> should be copied (this is the default value for <LENGTH>). For example, the following two calls to 'MidStr' result in 's' becoming 'four':

```
DEF s[30]:STRING
MidStr(s, 'Just four', 5)
MidStr(s, 'Just four apples', 5, 4)
```

'InStr(<STRING1>,<STRING2>,<STARTINDEX>=0)'

Returns the index of the first occurrence of <STRING2> in <STRING1> starting at <STARTINDEX> (in <STRING1>). <STARTINDEX> defaults to zero. If <STRING2> could not be found then -1 is returned.

'TrimStr(<STRING>)'

Returns the address of (i.e., a pointer to) the first non-white-space character in <STRING>. For instance, the following code fragment results in 's' becoming '12345'.

```
DEF s:PTR TO CHAR
s:=TrimStr('   \n \t       12345')
```

'LowerStr(<STRING>)'

Converts all uppercase letters in <STRING> to lowercase. This change is made "in-place", i.e., the contents of the string are directly affected. The string is returned for convenience.

'UpperStr(<STRING>)'

Converts all lowercase letters in <STRING> to uppercase. Again, this change is made in-place and the string is returned for convenience.

'SetStr(<E-STRING>,<LENGTH>)'

Sets the length of <E-STRING> to <LENGTH>. E-strings know how long they are, so if you alter an E-string (without using an E-string function) and change its size you need to set its length using this function before you can use it as an E-string again. For instance, if you've used an E-string like an array (which you can do) and written characters to it directly you must set its length before you can treat it as anything other than an array:

```
DEF s[10]:STRING s[0]:="a"
/* Remember that "a" is a character
value. */
s[1]:="b"
s[2]:="c"
/* At this point s is just an
array of CHAR. */
SetStr(s, 3)
/* Now, s can be used as an E-string >
again. */
SetStr(s, 2)
/* s is a bit shorter, but still an E-
string.*/
```

Notice that this function can be used to shorten an E-string, but this change is "destructive" (it cannot easily be reversed to give the original, longer E-string).

'Val(<STRING>,<ADDRESS>=NIL)'

What this function does is straight-forward but how you use it is a bit complicated. Basically, it converts <STRING> to a 'LONG' integer. Leading whitespace is ignored, and a leading '%' or '$' means that the string denotes a binary or hexadecimal integer (in the same way they do for numeric constants). The decoded integer is returned as the regular return value (*note Multiple Return Values::.). The number of characters of <STRING> that were read to make the integer is stored at <ADDRESS>, which is usually a variable address (from using '{<VAR>}'), and is also returned as the first optional return value. If <ADDRESS> is the special constant 'NIL' (i.e., zero) then this number is not stored (this is the default value for <ADDRESS>). You can use this number to calculate the position in the string which was not part of the integer. If an integer could not be decoded from the string then zero is returned as both return values and stored at <ADDRESS>.

Follow the comments in this example, and pay special attention to the use of the pointer 'p'.

```
DEF s[30]:STRING, value, chars, p:PTR
TO CHAR
StrCopy(s, ' \t \n 10 \t $3F -%0101010') >
value, chars:=Val('abcde 10 20')  -> Two >
return values...
/* After the above line, value and chars
will both be zero */
value:=Val(s, {chars}) -> Use address >
of chars
/* Value is now 10, chars will be 7 */
p:=s+chars
/* p now points to the space after the 10
in s */
value, chars:=Val(p)
/* Now value is $3F (63), chars is 6 */
```

Notice the two different ways of finding the number of characters read: a multiple-assignment and using the address of a variable.

There's a couple of other string functions ('ReadStr' and 'StringF') which will be discussed later.

8.5.3 Lists and E-lists

Lists are just like strings with 'LONG' elements rather than 'CHAR' elements (so they are very much like 'ARRAY OF LONG'). The list equivalent of an E-string is something called an "E-list". It has the same properties as an E-string, except the elements are 'LONG' (so could be pointers). Normal lists are most like string constants, except that the elements can be built from variables and so do not have to be constants. Just as strings are not true E-strings, (normal) lists are not true E-lists.

Lists are written using '[' and ']' to delimit comma separated elements. Like string constants a list returns the address of the memory which contains the elements.

For example the following code fragment:

```
DEF list:PTR TO LONG, number
number:=22
list:=[1,2,number]
```

is equivalent to:

```
DEF list[3]:ARRAY OF LONG, number
number:=22
list[0]:=1
list[1]:=2
list[2]:=number
```

Now, which of these two versions would you rather write? As you can see, lists are pretty useful for making your program easier to write and much easier to read.

E-list variables are like E-string variables and are declared in much the same way.The following code fragment declares 'lt' to be an E-list of maximum size 30. As ever, 'lt' is then a pointer (to 'LONG'), and it points to the memory allocated by the declaration.

```
DEF lt[30]:LIST
```

Lists are most useful for writing "tag lists", which are increasingly used in important Amiga system functions. A tag list is a list where the elements are thought of in pairs. The first

element of a pair is the tag, and the second is some data for that tag. See the 'Rom Kernel Reference Manual (Libraries)' for more details.

8.5.4 List functions

There are a number of list functions which are very similar to the string functions. Remember that E-lists are the list equivalents of E-strings, i.e. they can be altered and extended safely without exceeding their bounds. As with E-strings, E-lists are downwardly compatible with lists. Therefore, if a function requires a list as a parameter you can supply a list or an E-list. But if a function requires an E-list you cannot use a list in its place.

'List(<MAXSIZE>)'

Allocates memory for an E-list of maximum size <MAXSIZE> and returns a pointer to the list data. It is used to make space for a new E-list, like a 'LIST' declaration does. The following code fragments are (as with 'String') practically equivalent:

```
DEF lt[46]:LIST

DEF lt:PTR TO LONG
lt:=List(46)
```

You need to check that the return value from 'List' is not 'NIL' before you use it as an E-list. Like 'String', the memory allocated using 'List' is deallocated using 'DisposeLink'.

'ListCmp(<LIST1>,<LIST2>,<LENGTH>=ALL)'

Compares <LIST1> with <LIST2> (they can both be normal or E-lists). Works just like 'StrCmp' does for E-strings, so, for example, the following comparisons all return 'TRUE':

```
ListCmp([1,2,3,4],  [1,2,3,4])
ListCmp([1,2,3,4],  [1,2,3,7], 3)
```

'ListCopy(<E-LIST>,<LIST>,<LENGTH>=ALL)'

Works just like 'StrCopy', and the following example shows how to initialise an E-list:

```
DEF lt[7]:LIST, x
x:=4
ListCopy(lt, [1,2,3,x])
```

As with 'StrCopy', an E-list cannot be over-filled using 'ListCopy'.

'ListAdd(<E-LIST>,<LIST>,<LENGTH>=ALL)'

Works just like 'StrAdd', so this next code fragment results in the E-list 'lt' becoming the E-list version of '[1,2,3,4,5,6,7,8]'.

```
DEF lt[30]:LIST
ListCopy(lt, [1,2,3,4])
ListAdd(lt, [5,6,7,8])
```

'ListLen(<LIST>)'

Works just like 'StrLen', returning the length of <LIST>. There is no E-list specific length function.

'ListMax(<E-LIST>)'

Works just like 'StrMax', returning the maximum length of the <E-LIST>.

'SetList(<E-LIST>,<LENGTH>)'

Works just like 'SetStr', setting the length of <E-LIST> to <LENGTH>.

'ListItem(<LIST>,<INDEX>)'

Returns the element of <LIST> at <INDEX>. For example, if 'lt' is an E-list (so a 'PTR TO LONG') then 'ListItem(lt,n)' is the same as 'lt[n]'. This function is most useful when the list is not an E-list; for example, the following two code fragments are equivalent:

```
WriteF(ListItem(['Fred','Barney','Wilma'],
name))

DEF lt:PTR TO LONG
lt:=['Fred','Barney','Wilma']
WriteF(lt[name])
```

8.5.5 Complex types

In E the 'STRING' and 'LIST' types are called "complex" types. Complex-typed variables can also be created using the 'String' and 'List' functions as we've seen in the previous sections.

8.5.6 Typed lists

Normal lists contain 'LONG' elements, so you can write initialised arrays of 'LONG' elements. What about other kinds of array? Well, that's what "typed" lists are for. You specify the type of the elements of a list using ':<TYPE>' after the closing ']'. The allowable types are 'CHAR', 'INT', 'LONG' and any object type. There is a subtle difference between a normal, 'LONG' list and a typed list (even a 'LONG' typed list): only normal lists can be used with the list functions . For this reason, the term 'list' tends to refer only to normal lists.

The following code fragment uses the object 'rec' defined earlier and gives a couple of examples of typed lists:

```
DEF ints:PTR TO INT, objects:PTR TO rec,
p:PTR TO CHAR
ints:=[1,2,3]:INT
p:='fred'
objects:=[1,2,p,4, 300,301,'barney',303]:rec
```

It is equivalent to:

```
DEF ints[3]:ARRAY OF INT, objects[2]:ARRAY
OF rec, p:PTR TO CHAR ints[0]:=1
ints[1]:=2
ints[2]:=3
p:='fred'
objects[0].tag:=1
objects[0].check:=2
objects[0].table:=p
objects[0].data:=4
objects[1].table:='barney'
objects[1].tag:=300
objects[1].data:=303
objects[1].check:=301
```

The last group of assignments to 'objects[1]' have deliberately been shuffled in order to emphasise that the order of the elements in the "definition" of the object 'rec' is significant. Each of the elements of the list corresponds to an element in the object, and the order of elements in the list corresponds to the order in the object definition. In the example, the (object) list assignment line was broken after the end of the first object (the fourth element) to make it a bit more readable.

The last object in the list need not be completely defined, so, for instance, the second line of the assignment could have contained only three elements. This makes an object-typed list slightly different from the corresponding array of objects, since an array always defines a whole number of objects. With an object-typed list you must be careful not to access the undefined elements of a partially defined, trailing object.

Chapter 9

9. More About Statements and Expressions

This chapter details various E statements and expressions that were not covered in Part One. It also completes some of the partial descriptions given in Part One.

9.1 Initialised Declarations

Some variables can be initialised using constants in their declarations. The variables you cannot initialise in this way are array and complex type variables (and procedure parameters, obviously). All the other kinds can be initialised, whether they are local or global. An "initialised declaration" looks very much like a constant definition, with the value following the variable name and a '=' character joining them. The following example illustrates initialised declarations:

```
SET ENGLISH, FRENCH, GERMAN, JAPANESE,
RUSSIAN

CONST FREDLANGS=ENGLISH OR FRENCH OR GERMAN

DEF fredspeak=FREDLANGS,
   p=NIL:PTR TO LONG, q=0:PTR TO rec

PROC fred()
   DEF x=1, y=88
   /* Rest of procedure */
ENDPROC
```

Notice how you need to use a constant like 'FREDLANGS' in order to initialise the declaration of 'fredspeak' to something mildly complicated. Also, notice the initialisation of the point-

ers 'p' and 'q', and the position of the type information.

Of course, if you want to initialise variables with anything more than a simple constant you can use assignments at the start of the code. Generally, you should always initialise your variables (using either method) so that they are guaranteed to have a sensible value when you use them. Using the value of a variable that you haven't initialised in some way will probably get you in to a lot of trouble, because the value will just be something random that happened to be in the memory which is now being used by the variable. There are rules for how E initialises some kinds of variables (see the 'Reference Manual', but it's wise to explicitly initialise even those, as (strangely enough!) this will make your program more readable.

9.2 Assignments

We've already seen some assignments— these were assignment statements. Assignment expressions are similar except (as you've guessed) they can be used in expressions. This is because they return the value on the right-hand side of the assignment as well as performing the assignment. This is useful for efficiently checking that the value that's been assigned is sensible. For instance, the following code fragments are equivalent, but the first uses an assignment expression instead of a normal assignment statement.

```
IF (x:=y*z)=0
   WriteF('Error: y*z is zero (and x >
   is zero)\n')
ELSE
   WriteF('OK: y*z is not zero (and x >
   is y*z)\n')
ENDIF

x:=y*z
IF x=0
   WriteF('Error: y*z is zero (and x is >
   zero)\n')
ELSE
   WriteF('OK: y*z is not zero (and x is >
   y*z)\n')
ENDIF
```

You can easily tell the assignment expression: it's in parentheses and not on a line by itself. Notice the use of parentheses to group the assignment expression. Technically, the assignment operator has a very low precedence. Less technically, it will take as much as it can of the right-hand side to form the value to be assigned, so you need to use parentheses to stop 'x' getting the value '((y*z)=0)' (which will be 'TRUE' or 'FALSE', i.e., -1 or zero).

Assignment expressions, however, don't allow as rich a left-hand side as assignment statements. The only thing allowed on the left-hand side of an assignment expression is a variable name, whereas the statement form allows:

```
<VAR>
<VAR> [ <EXPRESSION> ]
<VAR> . <OBJ_ELEMENT_NAME>
^ <VAR>
```

(With as many repetitions of object element selection and/or array indexing as the elements' types allow.) Each of these may end with '++' or '—'. Therefore, the following are all valid assignments (the last two use assignment expressions):

```
x:=2
x--:=1
x[a*b]:=rubble
x.apple++:=3
x[].banana.basket[6]:=3+full(9)
x[].pear--:=fred(2,4)

x.pear:=(y:=2)
WriteF('x is now \d\n',
x:=1+(y:=(z:=fred(3,5)/2)*8))
```

You may be wondering what the '++' or '—' affect. Well, it's very simple: they only affect the VAR, which is 'x' in all of the assignment statements above. Notice that 'x[].pear—' is the same as 'x.pear—', for the same reasons mentioned earlier.

9.3 More Expressions

This section discusses side-effects, details two new operators ('BUT' and 'SIZEOF') and completes the description of the 'AND' and 'OR' operators.

9.3.1 Side-effects

If evaluating an expression causes the contents of variables to

change then that expression is said to have "side-effects". An assignment expression is a simple example of an expression with side-effects. Less obvious ones involve function calls with pointers to variables, where the function alters the data being pointed to.

Generally, expressions with side-effects should be avoided unless it is really obvious what is happening. This is because it can be difficult to find problems with your program's code if subtleties are buried in complicated expressions. On the other hand, side-effecting expressions are concise and often very elegant. They are also useful for "obfuscating" your code (i.e., making it difficult to understand—a form of copy protection!).

9.3.2 'BUT' expression

'BUT' is used to sequence two expressions. '<EXP1> BUT <EXP2>' evaluates <EXP1>, and then evaluates and returns the value of <EXP2>. This may not seem very useful at first sight, but if the first expression is an assignment it allows for a more general assignment expression. For example, the following code fragments are equivalent:

```
fred((x:=12*3) BUT x+y)

x:=12*3
fred(x+y)
```

Notice that parentheses need to be used around the assignment expression (in the first fragment) for the reasons given earlier (*note Assignments::).

9.3.3 Bitwise 'AND' and 'OR'

As hinted in the earlier chapters, the operators 'AND' and 'OR' are not simply logical operators. In fact, they are both

X	Y	X OR Y	X AND Y
1	1	1	1
1	0	1	0
0	1	1	0
0	0	0	0

bit-wise operators, where a "bit" is a binary digit (i.e., the zeroes or ones in the binary form of a number). To see how they work we should look at what happens to zeroes and ones, illustrated on the chart on page 100 (left).

Now, when you 'AND' or 'OR' two numbers the corresponding bits (binary digits) of the numbers are compared individually, according to the above table. So if 'x' were '%0111010' and 'y' were '%1010010' then 'x AND y' would be '%0010010' and 'x OR y' would be '%1111010':

```
    %0111010        %0111010
AND             OR
    %1010010        %1010010
------------------------------
    %0010010    %1111010
```

The numbers (in binary form) are lined up above each other, just like you do additions with normal numbers (i.e., starting with the right-hand digits, and maybe padding with zeroes on the left-hand side). The two bits in each column are 'AND'-ed or 'OR'-ed to give the result below the line.

So, how does this work for 'TRUE' and 'FALSE', and logic operations? Well, 'FALSE' is the number zero, so all the bits of 'FALSE' are zeroes, and 'TRUE' is -1, which has all 32 bits as ones (these numbers are 'LONG' so they are 32-bit quantities). So 'AND'-ing and 'OR'-ing these values always gives numbers which have all zero bits (i.e., 'FALSE') or all one bits (i.e., 'TRUE'), as appropriate. It's only when you start mixing numbers that aren't zero or -1 that you can muck up the logic. The non-zero numbers one and four are (by themselves) considered to be true, but '4 AND 1' is '%100 AND %001' which is zero (i.e., false). So when you use 'AND' as the logical operator it's not strictly true that all non-zero numbers represent true. 'OR' does not give such problems so all non-zero numbers are treated as true. Run this example to see why you should be careful:

```
PROC main()
   test(TRUE,   'TRUE\t\t')
   test(FALSE, 'FALSE\t\t')
   test(1,'1\t\t')
   test(4,'4\t\t')
```

```
      test(TRUE OR TRUE,'TRUE OR TRUE\t')
      test(TRUE AND TRUE, 'TRUE AND TRUE\t')
      test(1 OR 4, '1 OR 4\t\t')
      test(1 AND 4,'1 AND 4\t\t')
   ENDPROC

   PROC test(x, title)
      WriteF(title)
      WriteF(IF x THEN ' is TRUE\n' ELSE ' >
      is FALSE\n')
   ENDPROC
```

So, 'AND' and 'OR' are primarily bit-wise operators, but they can be used as logical operators under most circumstances, with zero representing false and all other numbers representing true. Care must be taken when using 'AND' with some pairs of non-zero numbers, since the bit-wise 'AND' of such numbers does not always give a non-zero (or true) result.

You can easily turn any value into a real truth value using the expression 'x<>FALSE', where 'x' represents the value to be converted. For example, this expression is true: '(1<>FALSE) AND (4<>FALSE)'.

9.3.4 'SIZEOF' expression

'SIZEOF' returns the size, in bytes (8-bits, like a 'CHAR'), of an OBJECT or a built-in type (like 'LONG'). This can be useful for determining storage requirements. For instance, the following code fragment prints the size of the object 'rec':

```
   OBJECT rec
      tag, check
      table[8]:ARRAY
      data:LONG
   ENDOBJECT

   PROC main()
      WriteF('Size of rec object is \d >
      bytes\n', SIZEOF rec)
   ENDPROC
```

You may think that 'SIZEOF' is unnecessary because you can easily calculate the size of an object just by looking at the sizes of the elements. Whilst this is generally true (it was for the 'rec' object), there is one thing to be careful about: alignment.

This means that 'ARRAY', 'INT', 'LONG' and object typed elements must start at an even memory address. Normally this isn't a problem, but if you have an odd number of consecutive 'CHAR' typed elements or an odd sized 'ARRAY OF CHAR', an extra, "pad" byte is introduced into the object so that the following element is aligned properly. For an 'ARRAY OF CHAR' this pad byte could be considered part of the array, so in effect this means array sizes are rounded up to the nearest even number. Otherwise, pad bytes are just an unusable part of an object, and their presence means the object size is not quite what you'd expect. Try the following program:

```
   OBJECT rec2
      tag, check
      table[7]:ARRAY
      data:LONG
   ENDOBJECT

   PROC main()
   WriteF('Size of rec2 object is \d
   bytes\n', SIZEOF rec2)
   ENDPROC
```

The only difference between the 'rec' and 'rec2' objects is that the array size is seven in 'rec2'. If you run the program you'll see that the size of the object has not changed. We might just as well have declared the 'table' element to be a slightly bigger array (i.e., have eight elements).

Chapter 10

10. E Built-In Constants, Variables and Functions

This chapter describes the constants, variables and functions which are built in to the E language. You can add more by using modules, but that's a more advanced topic.

10.1 Built-In Constants

We've already met several built-in constants. Here's the complete list:

'TRUE', 'FALSE'
: The boolean constants. As numbers, 'TRUE' is -1 and 'FALSE' is zero.

'NIL'
: The bad pointer value. Several functions produce this value for a pointer if an error occurred. As a number, 'NIL' is zero.

'ALL'
: Used with string and list functions to indicate that all the string or list is to be used. As a number, 'ALL' is -1.

'GADGETSIZE'
: The minimum number of bytes required to hold all the data for one gadget.

'OLDFILE', 'NEWFILE'

Used with 'Open' to open an old or new file. See the 'AmigaDOS Manual' for more details.

'STRLEN'

The length of the last string constant used. Remember that a string constant is something between '' characters, so, for example, the following program prints the string 's' and then its length:

```
PROC main()
  DEF s:PTR TO CHAR, len
  s:='12345678'
  len:=STRLEN
  WriteF(s)
  WriteF('\nis \d characters long\n',len)
ENDPROC
```

10.2 Built-In Variables

The following variables are built in to E and are called "system variables". They are global so can be accessed from any procedure.

'arg'

This is a string which contains the "command line" arguments passed your program when it was run (from the Shell or CLI). For instance, if your program were called 'fred' and you ran it like this:

```
fred file.txt "a big file" another
```

then 'arg' would the string:

```
file.txt "a big file" another
```

If you have AmigaDOS 2.0 (or greater) you can use the system routine 'ReadArgs' to parse the command line in a much more versatile way.

'wbmessage'

This contains 'NIL' if your program was started from the Shell/CLI, otherwise it's a pointer to the Workbench message which contains information about the icons selected when you started the program from Workbench. So, if you started the program from Workbench 'wbmessage' will not be

'NIL' and it will contain the Workbench arguments, but if you started the program from the Shell/CLI 'wbmessage' will be 'NIL' and the argments will be in 'arg' (or via 'ReadArgs').

'stdin', 'stdout', 'conout'

The 'stdin' and 'stdout' variables contain the standard input and output file handles.If your program was started from the Shell/CLI they will be filehandles on the Shell/CLI window (and 'conout' will be 'NIL'). However, if your program was started from Workbench these will both be 'NIL', and in this case the first call to 'WriteF' will open an output 'CON:' window and store the file handle for the window in 'stdout' and 'conout'. The file handle stored in 'conout' will be closed using 'Close' when the program terminates, so you can set up your own 'CON:' window or file for use by the output functions and it automatically closed.

'stdrast'

The raster port used by E built-in graphics functions such as 'Box' and 'Plot'. This can be changed so that these functions draw on different screens etc.

'dosbase', 'execbase', 'gfxbase', 'intuitionbase'

These are pointers to the appropriate library base, and are initialised by the E startup code, i.e., the Dos, Exec, Graphics and Intuition libraries are all opened by E so you don't need to do it yourself. These libraries are also automatically closed by E, so you shouldn't close them yourself. However, you must explicitly open and close all other Amiga system libraries that you want to use. The other library base variables are defined in the accompanying module.

10.3 Built-In Functions

There are many built-in functions in E. We've already seen a lot of string and list functions, and we've used 'WriteF' for printing. The remaining functions are, generally, simplifications of complex Amiga system functions, or E versions of support functions found in languages like C and Pascal.

To understand the graphics and Intuition support functions completely you really need to get something like the 'Rom Kernel Reference Manual (Libraries)'. However, if you don't want to do anything too complicated you should be able to get by.

10.3.1 Input and output functions

'WriteF(<STRING>,<PARAM1>,<PARAM2>,...)'

Writes a string to the standard output and returns the number of characters written. If place-holders are used in the string then the appropriate number of parameters must be supplied after the string in the order they are to be printed as part of the string. So far we've only met the '\d' place-holder for decimal numbers. The complete list is:

PLACE-HOLDER	PARAMETER TYPE	PRINTS
\c	Number	Character
\d	Number	Decimal number
\h	Number	Hexadecimal number
\s	String	String

So to print a string you use the '\s' place-holder in the string and supply the string (i.e., a 'PTR TO CHAR') as a parameter. Try the following program (remember '\a' prints an apostrophe character):

```
*Example
  PROC main()
    DEF s[30]:STRING
    StrCopy(s, 'Hello world', ALL)
    WriteF('The third element of s is >
    "\c"\n', s[2])
    WriteF('or \d (decimal)\n',   s[2])
    WriteF('or \h (hexadecimal)\n', s[2])
    WriteF('and s itself is \a\s\a\n', s)
  ENDPROC
```

You can control how the parameter is formatted in the '\d', '\h' and '\s' fields using another collection of special character sequences before the place-holder and size specifiers after it. If no size is specified the field will be as big as the data requires. A fixed field size can be specified using '[<NUMBER>]' after the place-holder. For strings you can also use the size specifier '(MIN,MAX)' which specifies the minimum and maximum sizes of the field. By default the data is right justified in the field and the left part of the field is filled, if necessary, with spaces. The following sequences before the place-holder can change this:

SEQUENCE	MEANING
\l	Left justify in field
\r	Right justify in field
\z	Set fill character to "0"

See how these formatting controls affect this example:

```
  PROC main()
    DEF s[30]:STRING
    StrCopy(s, 'Hello world', ALL)
    WriteF('The third element of s is >
    "\c"\n', s[2])
    WriteF('or \d[4] (decimal)\n', s[2])
    WriteF('or \z\h[4](hexadecimal) >
  \n',s[2])
  WriteF('\a\s[5]\a are the first five >
    elements of s \n', s)
  WriteF('and s in a very big field >
    \a\s[20]\a\n', s)
  WriteF('and s left justified in it >
    \a\l\s[20]\a\n', s)
  ENDPROC
```

'WriteF' uses the standard output, and this file handle is stored in the 'stdout' variable. If your program is started from Workbench this variable will contain 'NIL'. In this case, the first call to 'WriteF' will open a special output window and

put the file handle in the variables 'stdout' and 'conout', as outlined above.

'PrintF(<STRING>,<PARAM1>,<PARAM2>,...)'

'PrintF' works just like 'WriteF' except it uses the more efficient,buffered output routines only available if your Amiga is using Kickstart version 37 or greater (i.e., AmigaDOS 2.04 and above).

'StringF(<E-STRING>,<STRING>,<ARG1>,<ARG2>,...)'

The same as 'WriteF' except that the result is written to <E-STRING> instead of being printed. For example, the following code fragment sets 's' to '00123 is a' (since the E-string is not long enough for the whole string):

```
DEF s[10]:STRING
StringF(s, '\z\d[5] is a number', 123)
```

'Out(<FILEHANDLE>,<CHAR>)'

Outputs a single character, <CHAR>, to the file or console window denoted by <FILEHANDLE>, and returns -1 to indicate success (so any other return value means an error occurred). For instance, <FILEHANDLE> could be 'stdout', in which case the character is written to the standard output. (You need to make sure 'stdout' is not 'NIL', and you can do this by using a 'WriteF('')' call.) In general, you obtain a filehandle using the Amiga system function 'Open' from the 'dos.library'.

'Inp(<FILEHANDLE>)'

Reads and returns a single character from <FILEHANDLE>. If -1 is returned then the end of the file (EOF) was reached, or there was an error.

'ReadStr(<FILEHANDLE>,<E-STRING>)'

Reads a whole string from <FILEHANDLE> and returns -1 if EOF was reached or an error occurred. Characters are read up to a linefeed or the size of the string, whichever is sooner. Therefore, the resulting

string may be only a partial line. If -1 is returned then EOF was reached or an error occurred, and in either case the string so far is still valid. So, you still need to check the string even if -1 is returned. This will most commonly happen with files that do not of zero length) if nothing more had been read from the file when the error o EOF happened.

This next little program reads continually from its input until an error occurs or the user types 'quit'. It echoes the lines that it reads in uppercase. If you type a line longer than ten characters you'll see it reads it in more than one go. Because of the way normal console windows work, you need to type a return before a line gets read by the program (but this allows you to edit the line before the program sees it). If the program is started from Workbench then 'stdin' would be 'NIL', so 'WriteF('')' is used to force 'stdout' to be valid, and in this case it will be a new console window which can be used to accept input! (To make the compiled program into a Workbench program you simply need to create a tool icon for it. A quick way of doing this is to copy an existing tool's icon.)

```
PROC main()
  DEF s[10]:STRING, fh
  WriteF('')
  fh:=IF stdin THEN stdin ELSE stdout
  WHILE ReadStr(fh, s)<>-1
    UpperStr(s)
  EXIT StrCmp(s, 'QUIT', ALL)
    WriteF('Read: \a\s\a\n', s)
  ENDWHILE
  WriteF('Finished\n')
ENDPROC
```

'FileLength(<STRING>)'

Returns the length of the file named in <STRING>, or -1 if the file doesn't exist or an error occurred. Notice that you don't need to 'Open' the file or have a filehandle, you just supply the filename.

'SetStdIn(<FILEHANDLE>)'

Returns the value of 'stdin' before setting it to <FILEHANDLE>. Therefore, the following code fragments are equivalent:

```
oldstdin:=SetStdIn(newstdin)

oldstdin:=stdin
stdin:=newstdin
```

'SetStdOut(<FILEHANDLE>)'

Returns the value of 'stdout' before setting it to <FILEHANDLE>, and is otherwise just like 'SetStdIn'.

10.3.2 Intuition support functions

The functions in this section are simplified versions of Amiga system functions (in the Intuition library, as the title suggests). To make best use of them you are probably going to need something like the 'Rom Kernel Reference Manual (Libraries)', especially if you want to understand the Amiga specific things like IDCMP and raster ports.

The descriptions given here vary slightly in style from the previous descriptions. All function parameters can be expressions which represent numbers or addresses, as appropriate. Because many of the functions take several parameters they have been named (fairly descriptively) so they can be more easily referenced.

'OpenW(<X>,<Y>,<WID>,<HGT>,<IDCMP>,<WFLGS>,<TITLE>,<SCRN>,<SFLGS>,<GADS>,<TAGS>=NIL)

Opens and returns a pointer to a window with the supplied properties. If for some reason the window could not be opened 'NIL' is returned.

<X>, <Y>

The position on the screen where the window will appear.

<WID>, <HGT>

The width and height of the window.

<IDCMP>, <WFLGS>

The IDCMP and window specific flags.

<TITLE>

The window title (a string) which appears on the title bar of the window.

<SCRN>, <SFLGS>

The screen on which the window should open. If <SFLGS> is 1 the window will be opened on Workbench, and <SCRN> is ignored (so it can be 'NIL'). If <SFLGS> is '$F' (i.e., 15) the window will open on the custom screen pointed to by <SCRN> (which must then be valid). See 'OpenS' to see how to open a custom screen and get a screen pointer.

<GADS>

A pointer to a gadget list, or 'NIL' if you don't want any gadgets. These are not the standard window gadgets, since they are specified using the window flags. A gadget list can be created using the 'Gadget' function.

<TAGS>

A tag-list of other options available under Kickstart version 37 or greater. This can normally be omitted since it defaults to 'NIL'. See the 'Rom Kernel Reference Manual (Libraries)' for details about the available tags and their meanings.

There's not enough space to describe all the fine details about windows and IDCMP (see the 'Rom Kernel Reference Manual (Libraries)' for complete details), but a brief description in terms of flags might be useful. On the following page there's small table of common IDCMP flags:

IDCMP FLAG	VALUE
IDCMP_NEWSIZE	$2
IDCMP_REFRESHWINDOW	$4
IDCMP_MOUSEBUTTONS	$8
IDCMP_MOUSEMOVE	$10
IDCMP_GADGETDOWN	$20
IDCMP_GADGETUP	$40
IDCMP_MENUPICK	$100
IDCMP_CLOSEWINDOW	$200
IDCMP_RAWKEY	$400
IDCMP_DISKINSERTED	$8000
IDCMP_DISKREMOVED	$10000

Here's a table of useful window flags:

WINDOW FLAG	VALUE
WFLG_SIZEGADGET	$1
WFLG_DRAGBAR	$2
WFLG_DEPTHGADGET	$4
WFLG_CLOSEGADGET	$8
WFLG_SIZEBRIGHT	$10
WFLG_SIZEBBOTTOM	$20
WFLG_SMART_REFRESH	$0
WFLG_SIMPLE_REFRESH	$40
WFLG_SUPER_BITMAP	$80
WFLG_BACKDROP	$100
WFLG_REPORTMOUSE	$200
WFLG_GIMMEZEROZERO	$400
WFLG_BORDERLESS	$800
WFLG_ACTIVATE	$1000

All these flags are defined in the module 'intuition/intuition', so if you use that module you can use the constants rather than having to write the less descriptive value. Of course, you can always define your own constants for the values that you use.

You use the flags by 'OR'-ing the ones you want together, in a similar way to using sets. However, you should supply

only IDCMP flags as part of the <IDCMP> parameter, and you should supply only window flags as part of the <WFLGS> parameter. So, to get IDCMP messages when a disk is inserted and when the close gadget is clicked you specify both of the flags 'IDCMP_DISKINSERTED' and 'IDCMP_CLOSEWINDOW' for the <IDCMP> parameter, either by 'OR'-ing the constants or (less readably) by using the calculated value '$8200'.

Some of the window flags require some of IDCMP flags to be used as well, if an effect is to be complete. For example, if you want your window to have a close gadget (a standard window gadget) you need to use 'WFLG_CLOSEGADGET' as one of the window flags. If you want that gadget to be useful then you need to get an IDCMP message when the gadget is clicked. You therefore need to use 'IDCMP_CLOSEWINDOW' as one of the IDCMP flags. So the full effect requires both a window and an IDCMP flag (a gadget is pretty useless if you can't tell when it's been clicked).

If you only want to output text to a window (and maybe do some input from a window), it may be better to use a "console" window. These provide a text based input and output window, and are opened using the Dos library function 'Open' with the appropriate 'CON:' file name. See the 'AmigaDOS Manual' for more details about console windows.

'CloseW(<WINPTR>)'

Closes the window which is pointed to by <WINPTR>. It's safe to give 'NIL' for <WINPTR>, but in this case, of course, no window will be closed! The window pointer is usually a pointer returned by a matching call to 'OpenW'. You *must* remember to close any windows you may have opened before terminating your program.

'OpenS(<WID>,<HGT>,<DEPTH>,<SCRNRES>,<TITLE>,<TAGS>=NIL)'

Opens and returns a pointer to a custom screen with the supplied properties. If for some reason the screen could not be opened 'NIL' is returned.

<WID>, <HGT>

The width and height of the screen.

<DEPTH>

The depth of the screen, i.e., the number of bit-planes. This can be a number in the range 1-8 for AGA machines, or 1-6 for pre-AGA machines. A screen with depth 3 will be able to show 2 to the power 3 (i.e., 8) different colours, since it will have 2 to the power 3 different pens (or colour registers) available. You can set the colours of pens using the 'SetColour' function.

<SCRNRES>

The screen resolution flags.

<TITLE>

The screen title (a string) which appears on the title bar of the screen.

<TAGS>

A tag-list of other options available under Kickstart version 37 or greater. See the 'Rom Kernel Reference Manual (Libraries)' for more details.

The screen resolution flags control the screen mode. The following (common) values are taken from the module 'graphics/view'. You can, if you want, define your own constants for the values that you use. Either way it's best to use descriptive constants rather than directly using the values.

MODE FLAG	VALUE
V_LACE	$4
V_SUPERHIRES	$20
V_PFBA	$40
V_EXTRA_HALFBRITE	$80
V_DUALPF	$400
V_HAM	$800
V_HIRES	$8000

So, to get a hires, interlaced screen you specify both of the flags 'V_HIRES' and 'V_LACE', either by 'OR'-ing the constants or (less readably) by using calculated value '$8004'.

'CloseS(<SCRNPTR>)'

Closes the screen which is pointed to by <SCRNPTR>. It's safe to give 'NIL' for <SCRNPTR>, but in this case, of course, no screen will be closed! The screen pointer is usually a pointer returned by a matching call to 'OpenS'. You *must* remember to close any screens you may have opened before terminating your program. Also, you *must* close all windows that you opened on your screen before you can close the screen.

'Gadget(<BUF>,<GLIST>,<ID>,<FLAGS>,<X>,<Y>,<WIDTH>,<TEXT>)'

Creates a new gadget with the supplied properties and returns a pointer to the next position in the (memory) buffer that can be used for a gadget.

<BUF>

This is the memory buffer, i.e., a chunk of allocated memory. The best way of allocating this memory is to declare an array of size <N>*GADGETSIZE, where <N> is the number of gadgets which are going to be created. The first call to 'Gadget' will use the array as the buffer, and subsequent calls use the result of the previous call as the buffer (since this function returns the next free position in the buffer).

<GLIST>

This is a pointer to the gadget list that is being created, i.e., the array used as the buffer. When you create the first gadget in the list using an array 'a', this parameter should be 'NIL'. For all other gadgets in the list this parameter should be the array 'a'.

<ID>

A number which identifies the gadget. It is best to give a unique number for each gadget; that way you can easily identify them. This number is the only way you can identify which gadget has been clicked.

<FLAGS>

The type of gadget to be created. Zero represents a normal gadget, one a boolean gadget (a toggle) and three a boolean that starts selected.

<X>, <Y>

The position of the gadget, relative to the top, left-hand corner of the window.

<WIDTH>

The width of the gadget (in pixels, not characters).

<TEXT>

The text (a string) which will centred in the gadget, so the<WIDTH> must be big enough to hold this text.

Once a gadget list has been created by possibly several calls to this function the list can be passed as the <GADS> parameter to 'OpenW'.

'Mouse()'

Returns the state of the mouse buttons (including the middle mouse button if you have a three-button mouse). This is a set of flags, and the individual flag values are:

BUTTON PRESSED	VALUE
Left	%001
Right	%010
Middle	%100

So, if this function returns '%001' you know the left button is being pressed, and if it returns '%110' you know the middle and right buttons are both being pressed.

This mouse function is not strictly the proper way to do things. It is suggested you use this function only for small tests or demo-like programs. 'LeftMouse' and 'WaitLeftMouse' can be used to do things in a friendly way, but are restricted to seeing when the left mouse button is

pressed. More generally, the proper way of getting mouse details is to use the appropriate IDCMP flags for your window, wait for events (using 'WaitIMessage', for example) and decode the received information.

'MouseX(<WINPTR>)'

Returns the X coordinate of the mouse pointer, rela tive to the window pointed to by <WINPTR>. As above, this mouse function is not strictly the proper way to do things.

'MouseY(<WINPTR>)'

Returns the Y coordinate of the mouse pointer, rela tive to the window pointed to by <WINPTR>. As above, this mouse function is not strictly the proper way to do things.

'LeftMouse(<WINPTR>)'

Returns 'TRUE' if left mouse button has been clicked in the window pointed to by <WINPTR>, and 'FALSE' otherwise. In order for this to work sensibly the window must have the IDCMP flag 'IDCMP_MOUSEBUTTONS' set (see above). This function does things in a proper, Intuition-friendly manner and so is a good alternative to the 'Mouse' function.

'WaitIMessage(<WINPTR>)'

This function waits for a message from Intuition for the window pointed to by <WINPTR> and returns the class of the message (which is an IDCMP flag). If you did not specify any IDCMP flags when the win dow was opened, or the specified messages could never happen (e.g., you asked only for gadget mes sages and you have no gadgets), then this function may wait forever When you've got a message you can use the 'MsgXXX' functions to get some more information about the message. See the 'Rom Kernel Reference Manual (Libraries)' for more details on Intuition and IDCMP.

This function is basically equivalent to the following function, except that the 'MsgXXX' functions can also access the message data held in the variables 'code', 'qual' and 'iaddr'.

```
PROC waitimessage(win:PTR TO window)
  DEF port,msg:PTR TO
  intuimessage,class,code,qual,iaddr >
  port:=win.userport
  IF (msg:=GetMsg(port))=NIL
    REPEAT
      WaitPort(port)
    UNTIL (msg:=GetMsg(port))<>NIL
  ENDIF
  class:=msg.class
  code:=msg.code
  qual:=msg.qualifier
  iaddr:=msg.iaddress
  ReplyMsg(msg)
ENDPROC class
```

'MsgCode()'

Returns the 'code' part of the message returned by 'WaitIMessage'.

'MsgIaddr()'

Returns the 'iaddr' part of the message returned by 'WaitIMessage'.

'MsgQualifier()'

Returns the 'qual' part of the message returned by 'WaitIMessage'.

'WaitLeftMouse(<WINPTR>)'

This function waits for the left mouse button to be clicked in the window pointed to by <WINPTR>. It is advisable to have the IDCMP flag 'IDCMP_MOUSEBUTTONS' set for the window (see above).

This function does things in a proper, Intuition-friendly manner and so is a good alternative to the 'Mouse' function.

10.3.3 Graphics functions

The functions in this section use the standard raster port, the address of which is held in the variable 'stdrast'. Most of the time you don't need to worry about this because the E functions which open windows and screens set up this variable. So, by default, these functions affect the last window or screen opened. When you close a window or screen, 'stdrast' becomes 'NIL' and calls to these functions have no effect.

The descriptions in this section follow the same style as the previous section.

'Plot(<X>,<Y>,<PEN>=1)'

Plots a single point (<X>,<Y>) in the specified pen colour. The position is relative to the top, left-hand corner of the window or screen that is the current raster port (normally the last screen or window to be opened). The range of pen values available depend on the screen setup, but are at best 0-255 on AGA machines and 0-31 on pre-AGA machines. As a guide, the background colour is usually pen zero, and the main foreground colour is pen one (and this is the default pen). You can set the colours of pens using the 'SetColour' function.

'Line(<X1>,<Y1>,<X2>,<Y2>,<PEN>=1)'

Draws the line (<X1>,<Y1>) to (<X2>,<Y2>) in the specified pen colour.

'Box(<X1>,<Y1>,<X2>,<Y2>,<PEN>=1)'

Draws the (filled) box with vertices (<X1>,<Y1>), (<X2>,<Y1>), (<X1>,<Y2>) and (<X2>,<Y2>) in the specified pen colour.

'Colour(<FORE-PEN>,<BACK-PEN>=0)'

Sets the foreground and background pen colours. As mentioned above, the background colour is normally pen zero and the main foreground is pen one. You can change these defaults with this function, and if you stick to having the background pen as pen zero then calling this function with one argument changes just the foreground pen.

'TextF(<X>,<Y>,<FORMAT-STRING>,<ARG1>,<ARG2>,...)'

This works just like 'WriteF' except the resulting string is drawn on the current raster port (usually the last window or screen to be opened), starting at point (<X>,<Y>). Take care not to use any line-feed, carriage return, tab or escape characters in the string—they don't behave like they do in 'WriteF'.

'SetColour(<SCRNPTR>,<PEN>,<R>,<G>,<B>)'

Sets the colour of colour register <PEN> for the screen pointed to by <SCRNPTR> to be the appropriate RGB value (i.e., red value <R>, green value <G> and blue value <B>). The 'pen' can be anything up to 255, depending on the screen depth. Regardless of the chipset being used, <R>, <G> and <B> are taken from the range zero to 255, so 24-bit colours are always specified. In operation, though, the values are scaled to 12-bit colour for non-AGA machines.

'SetStdRast(<NEWRAST>)'

Returns the value of 'stdrast' before setting it to the new value. The following code fragments are equivalent:

```
oldstdrast:=SetStdRast(newstdrast)

oldstdrast:=stdrast
stdrast:=newstdrast
```

'SetTopaz(<SIZE>=8)'

Sets the text font for the current raster port to Topaz at the specified size, which defaults to the standard size eight.

10.3.4 Maths and logic functions

We've already seen the standard arithmetic operators. The addition, '+', and subtraction, '-', operators use full 32-bit integers, but, for efficiency, multiplication, '*', and division, '/', use restricted values. You can use '*' only to multiply 16-bit integers, and the result will be a 32-bit integer. Similarly, you can use '/' only to divide a 32-bit integer by a 16-bit integer, and the result will be a 16-bit integer. The restrictions do not affect most calculations, but if you really need to use all 32-bit integers (and you can cope with overflows, etc.) you can use the 'Mul' and 'Div' functions. 'Mul(a,b)' corresponds to 'a*b', and 'Div(a,b)' corresponds to 'a/b'.

We've also met the logic operators 'AND' and 'OR', which we know are really bit-wise operators. You can also use the functions 'And' and 'Or' to do exactly the same as 'AND' and 'OR' (respectively). So, for instance, 'And(a,b)' is the same as 'a AND b'. The reason for these functions is because there are 'Not' and 'Eor' (bit-wise) functions, too (and there aren't operators for these). 'Not(a)' swaps one and zero bits, so, for instance, 'Not(TRUE)' is 'FALSE' and 'Not(FALSE)' is 'TRUE'. 'Eor(a,b)' is the exclusive version of 'Or' and does almost the same, except that 'Eor(1,1)' is 0 whereas 'Or(1,1)' is 1 (and this extends to all the bits). So, basically, 'Eor' tells you which bits are different, or, logically, if the truth values are different. Therefore, 'Eor(TRUE,TRUE)' is 'FALSE' and 'Eor(TRUE,FALSE)' is 'TRUE'. Here's a collection of other functions related to maths, logic or numbers in general:

'Abs(<EXPRESSION>)'

Returns the absolute value of <EXPRESSION>. The absolute value of a number is that number without any minus sign (i.e., its the size of a number, disregarding its sign). So, 'Abs(9)' is 9, and 'Abs(-9)' is also 9.

'Sign(<EXPRESSION>)'

Returns the sign of <EXPRESSION>, which is the value one if it is (strictly) positive, -1 if it is (strictly) negative and zero if it is zero.

'Even(<EXPRESSION>)'

Returns 'TRUE' if <EXPRESSION> represents an even number, and 'FALSE' otherwise. Obviously, a number is either odd or even!

'Odd(<EXPRESSION>)'

Returns 'TRUE' if <EXPRESSION> represents an odd number, and 'FALSE' otherwise.

'Max(<EXP1>, <EXP2>)'

Returns the maximum of <EXP1> and <EXP2>.

'Min(<EXP1>, <EXP2>)'

Returns the minimum of <EXP1> and <EXP2>.

'Bounds(<EXP>, <MINEXP>, <MAXEXP>)'

Returns the value of <EXP> bounded to the limits <MINEXP> (minimum bound) and <MAXEXP> (maximum bound). That is, if <EXP> lies between the bounds then <EXP> is returned, but if it is less than <MINEXP> then <MINEXP> is returned or if it is greater than <MAXEXP> then <MAXEXP> is returned. This is useful for, say, constraining a calculated value to be a valid (integer) percentage (i.e., a value between zero and one hundred).

The following code fragments are equivalent:

```
y:=Bounds(x, min, max)

y:=IF x<min THEN min ELSE IF x>max
THEN max ELSE x
```

'Mod(<EXP1>,<EXP2>)'

Returns the 16-bit remainder (or modulus) of the division of the 32-bit <EXP1> by the 16-bit <EXP2> as the regular return value (*note Multiple Return Values::), and the 16-bit result of the division as the first optional return value. For example, the first assignment in the following code sets 'a' to 5 (since 26=(7*3)+5), 'b' to 3, 'c' to -5 and 'd' to -3. It is important to notice that if <EXP1> is negative then the modulus will also be negative. This is because of the way integer division works: it simply discards fractional parts rather rounding.

```
a,b:=Mod(26,7)
c,d:=Mod(-26,7)
```

'Rnd(<EXPRESSION>)'

Returns a random number in the range 0 to (n-1), where <EXPRESSION> represents the value n. These numbers are pseudo-random, so although you appear to get a random value from each call, the sequence of numbers you get will probably be the same each time you run your program. Before you use 'Rnd' for the first time in your program you should call it with a negative number. This decides the starting point for the pseudo-random numbers.

'RndQ(<EXPRESSION>)'

Returns a random 32-bit value, based on the seed <EXPRESSION>. This function is quicker than 'Rnd', but returns values in the 32-bit range, not a specified range. The seed value is used to select different sequences of pseudo-random numbers, and the first call to 'RndQ' should use a large value for the seed.

'Shl(<EXP1>,<EXP2>)'

Returns the value represented by <EXP1> shifted <EXP2> bits to the left. For example, 'Shl(%0001110,2)' is '%0111000' and 'Shl(%0001011,3)' is '%1011000'. Shifting a number one bit to the left is generally the same as multiplying it by two (although this isn't true when you shift large positive or large negative values). (The new bits shifted in at the right are always zeroes.)

'Shr(<EXP1>,<EXP2>)'

Returns the value represented by <EXP1> shifted <EXP2> bits to the right. For example, 'Shr(%0001110,2)' is '%0000011' and 'Shr(%1011010,3)' is '%0001011'. Shifting a number one bit to the right is generally the same as dividing it by two. (The new bits shifted in at the left are always zeroes.)

'Long(<ADDR>), Int(<ADDR>), Char(<ADDR>)'

Returns the 'LONG', 'INT' or 'CHAR' value at the address <ADDR>. These functions should be used only when setting up a pointer and dereferencing it in the normal way would make your program cluttered and less readable. Use of functions like these is

often called "peeking" memory (especially in dialects of the BASIC language).

'PutLong(<ADDR>,<EXP>), PutInt(<ADDR>,<EXP>), PutChar(<ADDR>,<EXP>)'

Writes the 'LONG', 'INT' or 'CHAR' value represented by <EXP> to the address <ADDR>. Again, these functions should be used only when really necessary. Use of functions like these is often called "poking" memory.

10.3.5 System support functions

'New(<BYTES>)'

Returns a pointer to a newly allocated chunk of memory, which is <BYTES> number of bytes. If the memory could not be allocated 'NIL' is returned. The memory is initialised to zero in each byte, and taken from any available store (Fast or Chip memory, in that order of preference). When you've finished with this memory you can use 'Dispose' to free it for use elsewhere in your program. You don't have to 'Dispose' with memory you allocated with 'New' because your program will automatically free it when it terminates. This is "not" true for memory allocated using the normal Amiga system routines.

'NewR(<BYTES>)'

The same as 'New' except that if the memory could not be allocated then the exception '"MEM"' is raised (and so, in this case, the function does not return).

'NewM(<BYTES>,<TYPE>)'

The same as 'NewR' except that the <TYPE> of memory (Fast or Chip) to be allocated can be specified using flags. The flags are defined in the module 'exec/memory'. See the 'Rom Kernel Reference Manual (Libraries)' for details about the system function 'AllocMem' which uses these flags in the same way.

As useful example, here's a small program which allocates some cleared (i.e., zeroed) Chip memory.

```
MODULE 'exec/memory'

PROC main()
   DEF m
   m:=NewM(20, MEMF_CHIP OR MEMF_CLEAR)
   WriteF('Allocation succeeded, m = )
   $\h\n', m)
EXCEPT
   IF exception="NEW" THEN )
   WriteF('Failed\n')
ENDPROC
```

'Dispose(<ADDRESS>)'

Used to free memory allocated with 'New', 'NewR' or 'NewM'. You should rarely need to use this function because the memory is automatically freed when the program terminates.

'DisposeLink(<COMPLEX>)'

Used to free the memory allocated with 'String' or 'List'. Again, you should rarely need to use this function because the memory is automatically freed when the program terminates.

'FastNew(<BYTES>)'

The same as 'NewR' except it uses a very fast, recycling method of allocating memory. The memory allocated using 'FastNew' is, as ever, deallocated automatically at the end of a program, and can be deallocated before then using 'FastDispose'. Note that only 'FastDispose' can be used and that it differs slightly from the 'Dispose' and 'DisposeLink' functions (you have to specify the number of bytes again when deallocating).

'FastDispose(<ADDRESS>,<BYTES>)'

Used to free the memory allocated using 'FastNew'. The <BYTES> parameter must be the same as the <BYTES> used when the memory was allocated with 'FastNew', but the benefit is much faster

allocation and deallocation, and generally more efficient use of memory.

'CleanUp(<EXPRESSION>=0)'

Terminates the program at this point, and does the normal things an E program does when it finishes. The value denoted by <EXPRESSION> is returned as the error code for the program. It is the replacement for the AmigaDOS 'Exit' routine which should *never* be used in an E program. This is the only safe way of terminating a program, other than reaching the (logical) end of the 'main' procedure (which is by far the most common way!).

'CtrlC()'

Returns 'TRUE' if control-C has been pressed since the last call, and 'FALSE' otherwise. This is really sensible only for programs started from the Shell/CLI.

'FreeStack()'

Returns the current amount of free stack space for the program. Only complicated programs need worry about things like stack. Recursion is the main thing that eats a lot of stack space.

'KickVersion(<EXPRESSION>)'

Returns 'TRUE' if your Kickstart revision is at least that given by <EXPRESSION>, and 'FALSE' otherwise. For instance, 'KickVersion(37)' checks whether you're running with Kickstart version 37 or greater (i.e., AmigaDOS 2.04 and above).

Where to find out more:
The authors' own documentation is present on the coverdisk. The first disk, called 'Amiga_E_3.1i', has a 'docs' directory which has an AmigaGuide document called 'E.Guide inside.'

For those with Internet access, there is a mailing list with many experienced Amiga E programmers participating, including the author. To join send an E-Mail to amigae-request@bkhouse.cts.com' with the word 'HELP' on a line of its own in the body of the message.

Also on the Internet, numerous Amiga E support archives exist on the Aminet. The drawer 'dev/e' holds all the support files. FTP the INDEX in the dev directory for the descriptions of all the files.

Those users with a modem but no access to the Internet can find all the Amiga E support files from the Aminet also on Rob Dale's DarkSide II BBS on 0181-771-9100.

SeaSoft Computing can supply the Amiga E support files on floppy disk for those without a modem or Internet access. Call them on 01903-850-378 and ask about their Amiga E support pack.

IMPORTANT: All information contained in this guide was accurate at the time of going to press (Oct 1995). CU Amiga Magazine cannot be held responsible for any changes to the addresses or contacts relating to Amiga E after this date.

AMIGA

Amiga E guide + free with CU Amiga Magazine December 1995

FREE 132 PAGE AMIGA E GUID

Amiga E

The new Amiga programming language

CU Amiga Magazine